NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FUZZ TESTING OF INDUSTRIAL NETWORK PROTOCOLS IN PROGRAMMABLE LOGIC CONTROLLERS

by

James J. Gormley III

December 2017

Thesis Advisor: Thuy D. Nguyen
Co-Advisor: Cynthia Irvine

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE				*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.				
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2017	**3. REPORT TYPE AND DATES COVERED** Master's thesis		
4. TITLE AND SUBTITLE FUZZ TESTING OF INDUSTRIAL NETWORK PROTOCOLS IN PROGRAMMABLE LOGIC CONTROLLERS				**5. FUNDING NUMBERS**
6. AUTHOR(S) James J. Gormley III				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000				**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A				**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.				**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Daily operations of U.S. Navy afloat and ashore systems are heavily reliant on industrial control systems (ICSs) to manage critical infrastructure services. Programmable logic controllers (PLCs) are vital components in these cyber-physical systems. The industrial network protocols used to communicate between nodes in a control network are complex and vulnerable to a myriad of cyber attacks, as reported by Department of Homeland Security Industrial Control Systems Cyber Emergency Response Team. This thesis utilizes protocol fuzz testing techniques to investigate potential vulnerabilities in the Allen-Bradley/Rockwell Automation (AB/RA) MicroLogix 1100 PLC through its implementation of EtherNet/IP, Common Industrial Protocol (CIP), and Programmable Controller Communication Commands (PCCC) communication protocols. This research also examines whether cross-generational vulnerabilities exist in the more advanced AB/RA ControlLogix 1756-L71 PLC. Our results discover several deviations from the EtherNet/IP and PCCC specifications in the MicroLogix 1100 implementation of these protocols. Additionally, we find that a recently disclosed denial-of-service vulnerability that renders the MicroLogix 1100 inoperable does not trigger a similar fault condition in the ControlLogix PLC.

14. SUBJECT TERMS industrial control system, protocol fuzz testing, PLC, EtherNet/IP, CIP, PCCC, MicroLogix, ControlLogix			**15. NUMBER OF PAGES** 175
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

FUZZ TESTING OF INDUSTRIAL NETWORK PROTOCOLS IN PROGRAMMABLE LOGIC CONTROLLERS

James J. Gormley III
Lieutenant Commander, United States Navy
B.S., Villanova University, 2005
M.P.S., The George Washington University, 2013

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN CYBER SYSTEMS AND OPERATIONS

from the

NAVAL POSTGRADUATE SCHOOL
December 2017

Approved by: Thuy D. Nguyen
Thesis Advisor

Dr. Cynthia Irvine
Co-Advisor

Dr. Dan Boger
Chair, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Daily operations of U.S. Navy afloat and ashore systems are heavily reliant on industrial control systems (ICSs) to manage critical infrastructure services. Programmable logic controllers (PLCs) are vital components in these cyber-physical systems. The industrial network protocols used to communicate between nodes in a control network are complex and vulnerable to a myriad of cyber attacks, as reported by Department of Homeland Security Industrial Control Systems Cyber Emergency Response Team. This thesis utilizes protocol fuzz testing techniques to investigate potential vulnerabilities in the Allen-Bradley/Rockwell Automation (AB/RA) MicroLogix 1100 PLC through its implementation of EtherNet/IP, Common Industrial Protocol (CIP), and Programmable Controller Communication Commands (PCCC) communication protocols. This research also examines whether cross-generational vulnerabilities exist in the more advanced AB/RA ControlLogix 1756-L71 PLC. Our results discover several deviations from the EtherNet/IP and PCCC specifications in the MicroLogix 1100 implementation of these protocols. Additionally, we find that a recently disclosed denial-of-service vulnerability that renders the MicroLogix 1100 inoperable does not trigger a similar fault condition in the ControlLogix PLC.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AB/RA	Allen Bradley/Rockwell Automation
CIP	Common Industrial Protocol
DNP3	Distributed Network Protocol
DoS	denial of service
EtherNet/IP	EtherNet Industrial Protocol
ENIP	EtherNet/IP
EXT STS	Extended Status
HM&E	hull mechanical and electrical
ICS	Industrial Control System
IOI	Internal Object Identifier
NOP	No Operation
ODVA	Open DeviceNet Vendor Association
PCCC	Programmable Controller Communication Commands
PROM	programmable read-only memory
PLC	programmable logic controller
RAM	random access memory
TCP	Transmission Control Protocol
SCADA	supervisory control and data acquisition
STS	Status
UDP	User Datagram Protocol

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my thesis advisors, Thuy Nguyen and Dr. Cynthia Irvine, for their guidance, expertise, and encouragement. You challenged me and drove me to make the most of this opportunity. I would also like to thank Francisco Tacliad, whose thesis laid the foundation for our research, and David Shifflet, for his assistance in creating our testing environment. To my wife, Ashley, thank you for your unwavering support. Your love, patience, and encouragement were invaluable during this process. To my daughter, Grace, and newborn son, James, you bring me laughter and joy every day and have been a source of great motivation. Thank you also to my parents and sister who continually support me, not just during this period, but throughout my life. Lastly, thank you to the Naval Postgraduate School professors, from whom I have learned so much over these past two years.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

While industrial control systems (ICSs) allow for the management of large, complex, and often distributed machinery systems, they can also be manipulated for malicious purposes. In 2000, a disgruntled former employee in Queensland, Australia, perpetrated one of the first known attacks on a Supervisory Control and Data Acquisition (SCADA) system. Through manipulation of the pumping stations, the offender released over one million liters of sewage into local waterways [1]. Following the events in Australia, attackers have successfully exploited the vulnerabilities inherent in networked control systems. The Stuxnet worm, discovered in 2010, targeted specific Siemens programmable logic controllers (PLCs) used at the Natanz nuclear enrichment facility. The sophisticated malware utilized four zero-day vulnerabilities to send fatigue-inducing commands to PLCs controlling nuclear enrichment centrifuges [2].

SCADA-based power grids are also vulnerable to cyber attacks. In December 2015, the Prykarpattyaoblenergo control center in Ukraine was the victim of an attack that left more than 230,000 West Ukraine residents without power for six hours. The alleged Russian attackers gained access to the utility's network through a phishing scheme. Using a program called BlackEnergy3, the hackers established a backdoor on the network, from which they gained access to the SCADA networks. The attackers were able to take thirty substations offline, disable backup power, and rewrite substation firmware before using KillDisk malware to delete files from operator systems and render them unusable [3].

Attacks against networked control systems can take varied forms. In April 2017, hackers simultaneously set off all 156 tornado warning sirens in Dallas, Texas. In normal operation, police dispatchers or weather officials send signals to a transmitter that activates selected sirens. To set off the sirens, the attacker used the input frequency to repeatedly activate all of the sirens over a period of several hours [4]. Although this attack relied on hijacking radio frequencies, similar disruption could potentially be caused by malicious software.

Efforts to address PLC vulnerabilities started several years ago. In 2007, the Idaho National Laboratory conducted the Aurora test in which researchers caused physical damage to a diesel generator by rapidly connecting and disconnecting the generator to the power grid, causing an out of phase condition [5]. In 2013, Sandia National Laboratories developed a system called Weaselboard, which provides zero-day exploit protection for PLCs by monitoring PLC backplane communications between devices and scanning for configuration changes [6]. In the private sector, Digital Bond created Project Basecamp to perform security testing on popular SCADA system components. The Project Basecamp researchers demonstrated vulnerabilities affecting multiple different PLC market leaders [7].

One effective method to test for vulnerabilities in protocols and systems is *fuzzing* or *fuzz testing*. Fuzzing is a technique that aims to uncover coding errors or security flaws by feeding a target program random input parameters [8]. Previous work has demonstrated that fuzz testing can be used to uncover vulnerabilities in industrial network protocol [9].

B. RELEVANCE TO THE NAVY

The Hull Mechanical and Electrical (HM&E) systems on U.S. Navy ships employ industrial automation components such as PLCs to run critical onboard services like propulsion, auxiliary, and mission-specific equipment [10]. As the Navy reduces shipboard crew strength through automation, as demonstrated in the DDG 1000, and launches completely unmanned vessels like the Anti-Submarine Warfare Continuous Trail Unmanned Vessel, the reliance on shipboard ICS increases. While a networked control system architecture (e.g., SCADA) provides centralized data availability and control of physical equipment in different locations, the communication channels between the PLCs and control devices are vulnerable to cyber attacks [2], [11].

Recognizing the inherent vulnerabilities in control systems, the Navy is developing the Resilient Hull, Mechanical, and Electrical Security system to prevent attackers from disabling or accessing shipboard PLCs. The system varies the implementation of PLC firmware so that if an exploit is able to disable a primary controller, the same exploit will not affect the redundant PLC's ability to assume the operation [12].

Allen Bradley / Rockwell Automation (AB/RA) is a leader in the ICS field and their products are currently used onboard Navy ships. Grandgenett et al. showed that AB/RA PLCs are susceptible to denial-of-service (DoS) [13], man-in-the-middle attacks, and replay attacks to force unauthorized privileged commands [14]. AB/RA PLCs support two widely-used industrial control protocols: Common Industrial Protocol (CIP) [15], EtherNet/IP (ENIP) [16], in addition to Programmable Controller Communication Commands (PCCC), a legacy AB/RA proprietary protocol [17]. CIP is an industry-vetted network protocol used to manage industrial devices [15]. CIP rides on top of ENIP, which is transported over TCP/IP. ICS network protocols, like CIP, ENIP, and PCCC allow for efficient control of distributed systems, but also create potential vectors of attack to disable or destroy U.S. Navy ships.

C. OBJECTIVES

This thesis aims to identify vulnerabilities in select AB/RA PLCs through their implementation of CIP and ENIP to directly improve mission readiness of U.S. Navy ships and harden their cyber defenses. Tacliad discusses the discovery of a CIP-encapsulated PCCC vulnerability in an AB/RA MicroLogix PLC through fuzz testing different ENIP, CIP, and PCCC commands [9]. This thesis seeks to expand and improve the ENIP Fuzz program to include additional ENIP, CIP, and PCCC commands. Once adapted to fuzz a larger catalogue of commands, we aim to implement ENIP Fuzz on the MicroLogix PLC and a more advanced AB/RA PLC (ControlLogix) to determine if vulnerabilities to AB/RA PLC communications stack are cross-generational.

D. THESIS ORGANIZATION

Chapter II provides background on CIP, EtherNet/IP, and PCCC protocols. It includes a summary of previous ICS fuzz testing efforts, Scapy [18], and existing Scapy-based fuzzing tools. Chapter II also presents an introduction to two AB/RA PLCs used in this thesis. Chapter III describes the experimentation design objectives, methodology, and testing environment. Chapter IV is an account of test plan and implementation. Chapter V is our analysis of results. Chapter VI discusses conclusions and future work.

II. BACKGROUND

A. COMMON INDUSTRIAL PROTOCOL (CIP)

CIP, previously known as Control and Information Protocol [19], is a “peer-to-peer object oriented protocol that provides connections between industrial devices (sensors, actuators) and higher-level devices (controllers)” [15]. CIP was developed by Rockwell Automation but is now run by Open DeviceNet Vendors Association (ODVA), a global association of automation industry leaders. CIP is supported by four different ODVA network communication protocols, EtherNet/IP, DeviceNet, CompoNet and ControlNet [20]. Using the Open System Interconnection model, CIP utilizes the Presentation and Application layers. Session layer is not utilized in CIP. In the EtherNet/IP structure, CIP rides on top of the Transport layer and utilizes an Ethernet network stack [21]. Figure 1 illustrates the CIP network work stack architecture.

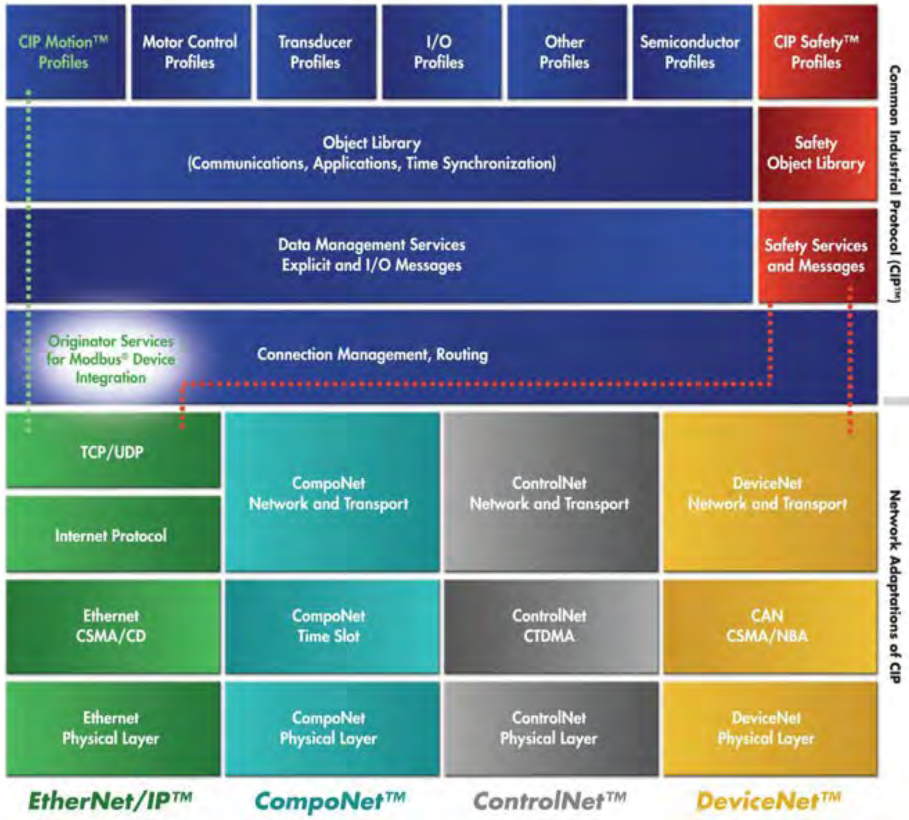


Figure 1. CIP Network Architecture Stack. Source: [15].

CIP nodes are comprised of *objects*, which can contain data. Each object is an *instance* of a particular *class*. CIP objects contain attributes for both object and class, which enable specific services. Objects with the same attributes belong to the same class [15]. CIP is designed so that the same objects on different devices behave in the same manner. This allows for a producer-consumer relationship, where data is sent from the producer device to potentially multiple consumer devices with a single transmission [22]. Figure 2 shows the CIP Object Model.

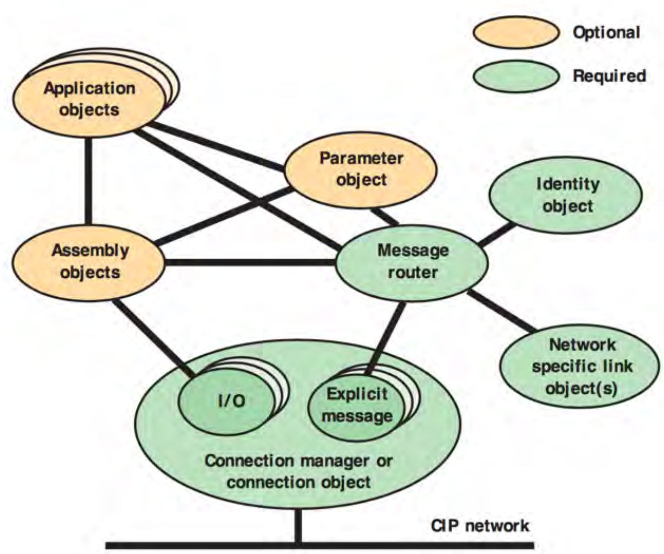


Figure 2. CIP Object Model. Source: [23].

CIP relies on two methods of routing to transmit data. For connected messages, CIP uses a connection ID to transfer packets. For unconnected messages, an Internal Object Identifier (IOI), also known as an EPATH, is used to explicitly provide the path packets will travel to their destination. The device that opens the connection dictates the routing directives [24].

B. ETHERNET/INDUSTRIAL PROTOCOL (ETHERNET/IP)

EtherNet/IP utilizes Ethernet (IEEE 802.3) and the TCP/IP network protocol stack to transport CIP as an application layer protocol. For this reason, it is often referred to as "CIP over Ethernet" [21]. EtherNet/IP uses IP Multicast to enable a producer-consumer exchange of information between a sending device and receiving devices [15]. By utilizing a common Ethernet protocol stack, EtherNet/IP allows CIP to be used across different CIP networks and enables Internet compatibility and remote control capability [21]. Figure 3 shows how an EtherNet/IP message is embedded in the TCP data payload.

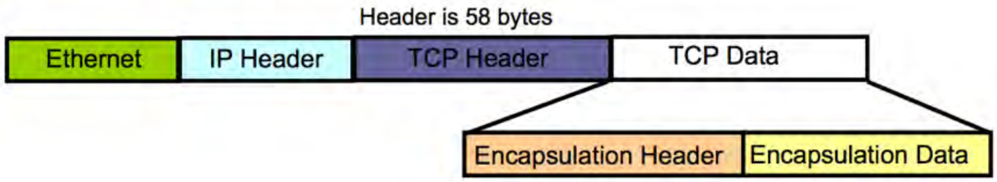


Figure 3. EtherNet/IP Packet Encapsulation. Source: [25].

The encapsulation message includes a standard 24-byte fixed length header, followed by an optional data section. Encapsulation messages may be in TCP or UDP format and are sent to port 44818 of the receiving device. Table 1 shows the content of the EtherNet/IP encapsulation header and encapsulated data [16].

Table 1. EtherNet/IP Packet Structure. Source: [16].

Structure	Field Name	Data Type	Field Value
Encapsulation header	Command	UINT	Encapsulation command
	Length	UINT	Length, in bytes, of the data portion of the message, i.e., the number of bytes following the header
	Session handle	UDINT	Session identification (application dependent)
	Status	UDINT	Status code
	Sender Context	ARRAY of octet	Information pertinent only to the sender of an encapsulation command. Length of 8.
	Options	UDINT	Options flags
Command specific data	Encapsulated data	ARRAY of 0 to 65511 octet	The encapsulation data portion of the message is required only for certain commands

C. PROGRAMMABLE CONTROLLER COMMUNICATION COMMANDS (PCCC)

PCCC is a legacy AB/RA protocol designed for the PLC5 and SLC500 processors [21]. PCCC objects do not support CIP connections on their own. However, they can be encapsulated in CIP commands in order to communicate with legacy PLCs. This encapsulation is accomplished through the use of an IOI. Once a connection to a Message Router object is established, an IOI is used to specify the PCCC object. When the CIP packet is received, “Execute PCCC” service is processed by the PCCC object at the receiving device [24]. Table 2 shows the message structure for a PCCC command, without CIP encapsulation [26].

Table 2. Message Format for Execute PCCC. Source: [26].

Request			Response		
Name	Data Type	Description	Name	Data Type	Description
Length	USINT	Length of requestor ID	Length	USINT	Length of requestor ID
Vendor	UINT	Vendor number of requestor	Vendor	UINT	Vendor number of requestor
Serial Number	UDINT	ASA serial number of requestor	Serial Number	UDINT	ASA serial number of requestor
Other	Product Specific	Identifier of user, task, etc. on the requestor	Other	Product Specific	Identifier of user, task, etc. on the requestor
CMD	USINT	Command byte	CMD	USINT	Command byte
STS	USINT	0	STS	USINT	Status byte
TNSW	UINT	Transport word	TNSW	UINT	Transport word. Same value as the request.
FNC	USINT	Function code. Not used for all CMD's.	EXT_STS	USINT	Extended status. Not used for all CMD's.
PCCC_params	ARRAY of USINT	CMD/FNC specific parameters	PCCC_results	ARRAY of USINT	CMD/FNC specific result data

D. FUZZ TESTING

The field of fuzz testing originated with Wisconsin University professor Barton Miller in 1989. Miller's team built a program, named *fuzz*, which generated random strings of characters and fed them into program inputs in an effort to create system failures [8]. Fuzz testing has grown into a widely-used method of vulnerability testing.

There are two main subcategories of fuzzers: generation-based and mutation. Generation-based fuzzers craft fuzzing inputs based on knowledge of input structures and protocols. These programs generate strings of random characters and varying lengths. Sophisticated generation-based fuzzers utilize block-based methods, where each input field is treated as a targetable fuzzing block [27]. These fuzzers require detailed specifications of input fields and protocols in order to customize block-sized inputs [28].

Mutation fuzzers utilize known good inputs and network traffic to build fuzzing structures. By taking the known good input and switching out acceptable values with random values, mutation fuzzers increase the likelihood their malformed inputs will not be rejected outright, which increases their effectiveness [27].

E. ICS FUZZERS

Numerous fuzz testing suites targeting well known ICS protocols are available. beSTORM offers a commercially available EtherNet/IP fuzzing tool [29]. Mu Test Suite, also a commercial product, includes resources to fuzz Distributed Network Protocol (DNP3), Modbus, and the IEC61850 protocol [27]. In the open source arena, the Sulley fuzzer includes modules for popular ICS protocols such as DNP3, Inter-Control Center Communications Protocol, and Modbus [30]. Developed at Dartmouth, LZFuzz fuzzes SCADA communications with unknown protocol structures. LZFuzz inserts itself into live traffic and captures packets. Packets inbound to the target are tokenized and sent though a mutation fuzzer to generate fuzzing inputs to the target. The program then monitors return traffic to the traffic source for indications of success [27].

This thesis research utilizes Tacliad's open source fuzzing tool, called ENIP Fuzz. ENIP Fuzz is an ICS fuzzing program that uses the Python-based packet manipulation tool, Scapy [18] to craft customized fuzzing inputs. ENIP Fuzz targets fields within ENIP and CIP request packets [9].

F. SCAPY

Scapy is a Python-based packet manipulation tool that can enable network probes and attacks. Scapy is flexible enough to allow custom packet crafting. It does not place limits on type of field input or stack configuration, which makes it a powerful tool for protocol fuzz testing. Users can craft Scapy packets in stackable layers. Scapy is capable of both sending and listening for response packets. Many networking tools apply interpretive filters on packet responses. Scapy does not employ this method in order to avoid inserting potential bias into response results. Interpretation of Scapy response packets lies with the user [18].

G. PREVIOUS SCAPY-BASED FUZZING

Scapy's versatile configuration has made it a popular choice for fuzz testing frameworks. Scapy allows a user to specify designated fields for fuzzing, while providing standard protocol inputs to other fields [31]. Scapy libraries have been used to fuzz Wi-Fi drivers [32], IPV6 [33] and IPV6 over low power wireless personal area networks [34], and Internet Key Exchange messages [35]. In the ICS field, different fuzzing tools have utilized Scapy. Modbus/TCP Fuzzer targets the Modbus communication protocol [36]. Modbus is an application layer protocol that utilizes a master-slave architecture [37]. Scapy is used to target the Modbus/TCP master-initiated command packets for fuzzing. Some electrical utilities use the IEEE C37.118 protocol to communicate between wide area monitoring systems that operate phasor measurement units and phasor data concentrators. Sprabery et al. created a IEEE C37.118 mutation-based fuzzer using Scapy to test particular protocol rules for vulnerabilities [38].

Tacliad’s ENIP Fuzz targets the EtherNet/IP and CIP protocols using the Scapy library to craft malformed packets. ENIP Fuzz tests specified objects in the designated protocols and monitors for unexpected responses or lack of response to liveliness checks. While Tacliad tested a very limited sample of EtherNet/IP, CIP, and CIP-encapsulated PCCC commands, his experimentation demonstrated a proof of concept, which can be greatly expanded to determine the robustness of the examined protocols [9].

H. ALLEN-BRADLEY / ROCKWELL AUTOMATION PLCS (MICROLOGIX 1100 AND CONGROLLOGIX 5570)

The MicroLogix 1100 is a lower-end PLC that supports 12 inputs (10 digital and 2 analog) and 6 outputs, and up to 144 digital I/O points. It is utilized to perform varied industrial applications such as machinery control and production processes. The controller has an RS232/485 serial port and an Ethernet port. The Ethernet port enables peer-to-peer communication across controllers [39]. Figure 4 shows a MicroLogix 1100.

Figure 4. MicroLogix 1100 PLC. Source: [39].

The AB/RA ControlLogix PLC is a more advanced modular PLC than the MicroLogix 1100. A ControlLogix PLC consists of a controller (CPU) module (e.g., 1756-L71 controller) and multiple I/O modules in one chassis. The local I/O modules can include one or more EtherNet/IP modules (e.g., 1756-EN2T and 1756-EWEB modules), and one or more analog and digital I/O modules (e.g., 1756-OF8 and 1756-IB16 modules). A ControlLogix 5570 PLC can handle up to 128,000 digital or 4,000 analog I/O points and is used for shipboard applications, power generation, and transportation functions. The PLC can communicate across multiple protocols including EtherNet/IP (including CIP and encapsulated PCCC), ControlNet, DeviceNet, Data Highway Plus, Remote I/O, SynchLink, and third-party networks. The 5570 model does not offer an embedded Ethernet Port, but has a USB interface for local programming. For ease of configuration and maintenance, most EtherNet/IP modules support web browsing, email, and file transfer. The ControlLogix family also offers the ability to configure controller redundancy into the system. [40]. Figure 5 shows a ControlLogix PLC with multiple I/O modules.

Figure 5. ControlLogix PLC. Source: [40].

III. DESIGN

A. OBJECTIVES

This thesis explores two objectives. The first objective is to determine if ENIP Fuzz can be used to determine new vulnerabilities in the AB/RA implementation of the ENIP, CIP and PCCC protocols used by the MicroLogix and ControlLogix PLCs. Our hypothesis is that undiscovered software flaws could potentially exist in the implementation of AB/RA's implementation of the protocols. The second objective is to determine if testing network vulnerabilities known to exist in older PLCs help inform on the robustness of the ICS network stack in a more modern PLC design. Our hypothesis is that legacy protocol handlers are left in the code base but not fully tested in newer PLC models.

B. METHODOLOGY

Testing follows a black box-style fuzzing methodology, i.e., having no access to AB/RA source code. The test plan and testing methodology relies heavily on the protocol specifications for ENIP, CIP, and PCCC protocols. To determine specific commands from each protocol to fuzz, we analyze protocol commands to identify targets that focus on non-disruptive functionality. We avoid commands that we assessed to have high risk of reconfiguring memory, altering functionality, or causing permanent damage to the SUT. We aim to select target commands that provide a representative sample of different types of services provided by each protocol.

Previous testing using ENIP Fuzz exercised three MicroLogix-supported commands sent over a TCP connection: ENIP Register Session, CIP No_Operation (NOP), and PCCC Execute Services [9]. Our testing framework focuses on a wider cross-section of ENIP commands and CIP services, transported over both TCP and UDP, in an effort to discover vulnerabilities that may be present in different service types.

The ENIP test commands can be grouped into five categories as shown in Table 3. Our ENIP test suite consists of all three "list" commands, the UnRegisterSession command, the SendRRData and SendUnitData commands, the reserved for legacy commands, and the reserved for future expansion commands.

Table 3. ENIP Test Commands. Source: [16].

ENIP Test Commands	Description
Lists	
List Identity	Requests information on the target's identity.
List Interfaces	Requests non-CIP communication interfaces associated with the target.
List Services	Requests information on the supported services.
Session Commands	
Unregister Session	Instructs the receiver to initiate a close of the underlying TCP/IP connection.
Send Commands	
SendRRData	Transfers an encapsulated request/reply packet.
SendUnitData	Sends encapsulated connected messages.
Legacy Commands	
Reserved Command Codes	Reserved for legacy use.
Future Expansion Commands	
Reserved Command Codes	Reserved for future expansion.

For the CIP Explicit Messaging testing, we select services with multiple fuzzable fields based on the assumption that such commands would be more complex and have a higher potential for vulnerabilities in handling errors. Table 4 summarizes the CIP common services in the CIP test suite. While each of the Get_Attributes_xxx services have a corresponding Set_Attributes_xxx command, we specifically skip the latter in an effort to not corrupt any PLC settings.

Table 4. CIP Test Commands. Source: [15].

CIP Test Commands	Description
Get Attribute All	Returns the contents of the instance or class attributes defined in the object definition.
Get Attribute List	Returns the contents of the selected gettable attributes of the specified object class or instance.
Get Attribute Single	Returns the contents of the specified attribute.
Find Next Object Instance	Returns a list of Instance IDs [15] associated with existing Object Instances [15]. Existing Objects are those that are currently accessible from the CIP subnet.

Our strategy for testing PCCC commands follows two common testing techniques: specification compliance testing and unexpected exception handling testing. First, we identify the PCCC commands that are described in the DF1 Protocol and Command Set specification [17] as compatible with the MicroLogix 1000 family's implementation of the protocol. PCCC information for the SUTs is not publicly available. Table 5 shows the commands in the PCCC test suite that have a low risk of disrupting the SUT functionality. We choose the PCCC Echo command because it allows the inclusion of a large amount of data in a packet, which can be used to test the maximum allowable packet size. We select the Protected Typed File Read, Protected Typed File Write, and Protected Logical Write with Three Address Fields commands for their multiple fuzzable fields and potential for stack corruption. The Unprotected Read command is selected for its potential to cause errors by attempting to read unintended address spaces. The Read Diagnostic Counters command is included in the test suite due to its ability to read data from a fuzzable address location. The Diagnostic Status command is also tested because the response to the Diagnostic Status request command provides the starting memory address for the PLC's diagnostic counters, which can be used with the Read Diagnostic Counters command.

In addition to the MicroLogix-supported PCCC commands, the PCCC test suite also includes commands that may contain vulnerabilities or cause an unexpected result because, according to the PCCC specification [17], they are not supported by the MicroLogix 1000 PLC (see Table 5). While the selected commands, Download Completed and Restart, do not have fuzzable fields, their inclusion in the test suite allows testing of unexpected error handling.

Table 5. PCCC Test Commands. Source: [17].

PCCC Test Commands	Description
Echo	The receiving module should reply to this command by transmitting the same data back to the originating node.
Protected Typed File Read	Reads data from an open file in the PLC.
Protected Typed File Write	Writes data to an open file in the PLC.
Protected Logical Write with Three Address Fields	Writes data to a logical address in PLC processor.
Unprotected Read	Read data from a common interface file.
Diagnostic Status	Reads a block of status information from an interface module.
Read Diagnostic Counters	Reads up to 244 bytes of data from the PROM or RAM of an interface module.
Restart	Revokes upload and download privileges for the source computer node and initializes PLC restart. (Command intended for PLC-3 only after completion of upload or download operation)
Download Completed	Places processor back in previous mode upon completion of system download.
Protected Typed Logical Read with Three Address Fields	Reads data from a logical address in PLC processor.

Previous ENIP Fuzz testing uncovered an improper input validation vulnerability in different versions of MicroLogix 1100 controllers, which is described in the ICS-CERT security advisory ICSA-17-138-03 [41]. When the Protected Typed Logical Read with Three Address Fields command was issued with certain parameters, the MicroLogix 1100 halted, causing a denial of service condition. This command is tested on a ControlLogix 5570 to verify our second hypothesis that legacy protocol handlers may be left in the code base but not fully tested in newer PLC models.

C. TEST ENVIRONMENT

The fuzzing tool used in this thesis is ENIP Fuzz. It is a Scapy-based fuzzer that enables construction of specially crafted packets, which allows the user to test a wide variety of inputs for each value in protocol packet. ENIP Fuzz utilizes both CIP and ENIP dissectors, which define classes for each protocol request and response message format.

The MicroLogix test environment consists of a MicroLogix system under test (SUT), a Windows PC with a Windows 7 virtual machine (VM), a Mac laptop with a Kali Linux 2.0 VM, and a Mac laptop running the Wireshark protocol analyzer. All components are connected to a central hub. The Windows 7 VM runs RSLinx and RSLogix—AB/RA development software with which a user can send commands to and monitor responses from the connected PLC. In the Kali VM, ENIP Fuzz is used to build and send custom packets to the PCL in order to test the ENIP, CIP, and PCCC protocols for vulnerabilities. During testing, potential faults are monitored on the RSLogix console, from fault responses in Wireshark, and physical fault indications on the SUT. The testing environment setup is displayed in Figure 6.

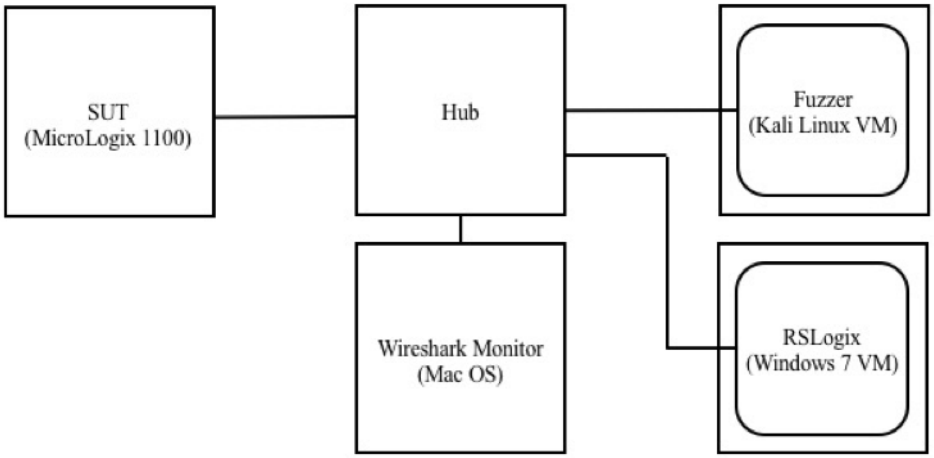


Figure 6. MicroLogix Testing Environment

The ControlLogix test environment is similar to the MicroLogix environment except that the Rockwell Studio 5000 Logic Designer development software running on a Window 7 PC is used instead of the RSLogix software (see Figure 7).

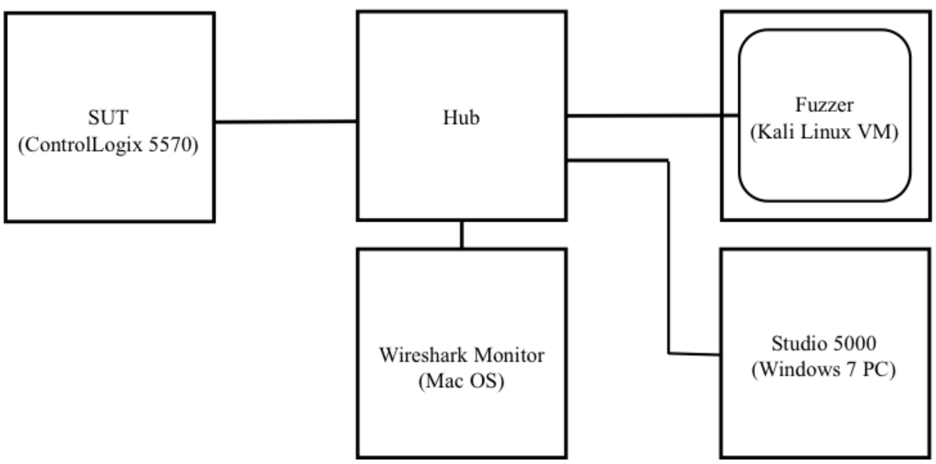


Figure 7. ControlLogix Testing Environment

IV. IMPLEMENTATION AND TEST PLAN

A. FUZZER IMPLEMENTATION

The fuzzing platform, ENIP Fuzz [9], is modified to conduct the desired breadth of target command testing across the ENIP, CIP, and PCCC protocols. Using the modified ENIP Fuzz program, properly formed packets are crafted and sent to the SUT to establish baseline request and response behavior. Specially designed malformed packets are then sent to the SUT and analyzed in relation to the hypothesized SUT responses. The testing goal is to trigger a denial of service condition in the SUT. This is defined as a fault in the SUT that requires either a power cycle to clear or reset through the RSLogix/Studio 5000 interfaces, or a disruption in the SUT's ability to send or receive command traffic.

1. FUZZER MODIFICATIONS FOR MICROLOGIX

The ENIP Fuzz architecture consists of command and service-specific fuzzing modules and protocol dissectors. Eight ENIP fuzzing modules are constructed to test the following ENIP commands (discussed in Chapter III): ListServices, ListIdentity, ListInterfaces, UnRegisterSession, SendRRData, SendUnitData, Reserved for Legacy Use, and Reserved for Future Use. Two CIP fuzzing modules are created to test the Get_Attributes_All and Find_Next_Object_Instance CIP services. Nine PCCC fuzzing modules are added to test the following PCCC commands via the PCCC Execute Service Request service: Echo, Protected Typed File Read, Protected Typed File Write, Protected Typed Logical Write with Three Address Fields, Unprotected Read, Download Completed, Restart, Diagnostic Status, and Read Diagnostic Counters.

The ENIP Fuzz CIP dissector is modified to allow the Find_Next_Object_Instance command to specify the number of maximum values returned. Both ENIP and CIP dissectors are modified to create the expanded packet views presented later in this document.

2. FUZZER MODIFICATIONS FOR CONTROLLOGIX

In order to test a recently discovered PCCC vulnerability [9] affecting MicroLogix on the ControlLogix PLC, ENIP Fuzz's handling of the Protected Typed Logical Read with Three Address Fields PCCC command requires modifications. The objective of this test is to determine whether the PCCC vulnerability in the MicroLogix implementation also exists in the ControlLogix software. Through analysis of ControlLogix network traffic, it is observed that the ControlLogix implements the CIP Forward_Open request differently. The Forward_Open request establishes a connection with a target device [15] and precedes the target test command request. ControlLogix PLCs require a 3-word request path [15], as opposed to the 2-word request path used on the MicroLogix. The request path specifies the required route the command packet travels to the remote target device [15]. ENIP Fuzz is modified to handle both types of request path.

B. ENIP FUZZING TEST PLAN

Previous ENIP Fuzz testing is limited to the RegisterSession command [9]. The current work expands the testing to test ENIP commands not tested by Tacliad [9] for vulnerabilities. Command fields are tested in isolation in order to provide a methodical evaluation of each command's potential vulnerabilities. Table 6 summarizes the ENIP test plan.

Table 6. ENIP Test Plan

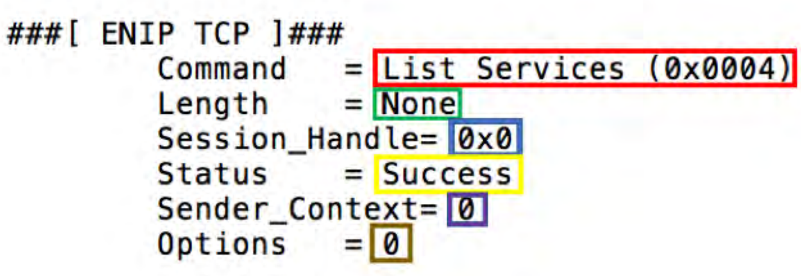
Test Number	ENIP Command	Fuzzed Field	Protocol	Fuzzing Parameters
T1	List Services/Identity/Interfaces	Session Handle	TCP	0x00000000 to 0xFFFFFFFF
T2	List Services/Identity/Interfaces	Session Handle	UDP	0x00000000 to 0xFFFFFFFF
T3	List Services/Identity/Interfaces	Status	TCP	0x00000000 to 0xFFFFFFFF
T4	List Services/Identity/Interfaces	Status	UDP	0x00000000 to 0xFFFFFFFF

Test Number	ENIP Command	Fuzzed Field	Protocol	Fuzzing Parameters
T5	List Services/Identity/Interfaces	Sender Context	TCP	0x0000000000000000 to 0xFFFFFFFFFFFFFFFF
T6	List Services/Identity/Interfaces	Sender Context	UDP	0x0000000000000000 to 0xFFFFFFFFFFFFFFFF
T7	List Services/Identity/Interfaces	Options	TCP	0x00000000 and 0xFFFFFFFF
T8	List Services/Identity/Interfaces	Options	UDP	0x00000000 and 0xFFFFFFFF
T9	UnRegisterSession	Session Handle	TCP	0x00000000 to 0xFFFFFFFF
T10	UnRegisterSession	Status	TCP	0x00000000 to 0xFFFFFFFF
T11	UnRegisterSession	Sender Context	TCP	0x0000000000000000 to 0xFFFFFFFFFFFFFFFF
T12	UnRegisterSession	Options	TCP	0x00000000 and 0xFFFFFFFF
T13	UnRegisterSession UDP Functionality	N/A	UDP	Properly crafted ENIP encapsulated packet sent over UDP
T14	SendRRData	Session Handle	TCP	0x00000000 to 0xFFFFFFFF
T15	SendRRData	Status	TCP	0x00000000 to 0xFFFFFFFF
T16	SendRRData	Sender Context	TCP	0x0000000000000000 to 0xFFFFFFFFFFFFFFFF
T17	SendRRData	Options	TCP	0x00000000 and 0xFFFFFFFF
T18	SendRRData	Interface Handle	TCP	0x00000000 and 0xFFFFFFFF
T19	SendRRData	TimeOut	TCP	0-65535
T20	SendUnitData	Session Handle	TCP	0x00000000 to 0xFFFFFFFF
T21	SendUnitData	Status	TCP	0x00000000 to 0xFFFFFFFF
T22	SendUnitData	Sender Context	TCP	0x0000000000000000 to 0xFFFFFFFFFFFFFFFF
T23	SendUnitData	Options	TCP	0x00000000 and 0xFFFFFFFF
T24	SendUnitData	Interface Handle	TCP	0x00000000 and 0xFFFFFFFF

Test Number	ENIP Command	Fuzzed Field	Protocol	Fuzzing Parameters
T25	SendUnitData	TimeOut	TCP	0-65535
T26	Reserved for Legacy	Command Field	TCP	0x0001,0x0002, 0x0005, 0x0067-0x006E, and 0x0071-0x00C7
T27	Reserved for Legacy	Command Field	UDP	0x0001,0x0002, 0x0005, 0x0067-0x006E, and 0x0071-0x00C7
T28	Reserved for Future Use	Command Field	TCP	0x0006-0x0062 and 0x00C8-0xFFFF
T29	Reserved for Future Use	Command Field	UDP	0x0006-0x0062 and 0x00C8-0xFFFF

1. ENIP ListServices Command

The ENIP ListServices Request command returns the service(s) the target supports. To test the command, the Session Handle, Status, Sender Context, and Options fields are individually fuzzed using both TCP and UDP. The Session Handle field is tested with a combination of inputs ranging from 0x00000000 to 0xFFFFFFFF. The Status field is fuzzed in a similar manner. The Sender Context field is tested with data ranging from 0x0000000000000000 to 0xFFFFFFFFFFFFFFFF. The Options field is tested between 0x00000000 and 0xFFFFFFFF. Figures 8–11 illustrate the packet structure for the ListServices command sent over TCP and UDP, respectively.



Fields encapsulated at the ENIP layer are highlighted.

Figure 8. An Example ENIP ListServices Request over TCP Packet

```
0000   45 00 00 40 00 01 00 00 40 06 f8 ed c0 a8 00 3e
0010   c0 a8 00 3b af 12 af 12 00 00 00 00 00 00 00 00
0020   50 02 20 00 ab db 00 00 04 00 00 00 00 00 00 00
0030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 9. Hexadecimal View of Example ENIP ListServices Request over TCP Packet

```
###[ ENIP UDP ]###
        Command   = List Services (0x0004)
        Length    = None
        Session_Handle= 0x0
        Status    = Success
        Sender_Context= 0
        Options   = 0
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 10. An Example ENIP ListServices Request over UDP Packet

```
0000   45 00 00 34 00 01 00 00 40 11 f8 ee c0 a8 00 3e
0010   c0 a8 00 3b f3 7a af 12 00 20 d7 56 04 00 00 00
0020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030   00 00 00 00
```

Figure 11. Hexadecimal View of Example ENIP ListServices Request over UDP Packet

2. ENIP UnRegisterSession Command

The ENIP UnRegisterSession command terminates an existing ENIP session and closes the TCP connection associated with the particular ENIP session. An ENIP session is established using the ENIP RegisterSession command that was previously tested [9]. After receiving the UnRegisterSession command, the receiver initiates the closing of the TCP connection and does not reply with a response message. In the event this command is sent via UDP, the receiver replies with an error code 0x01, indicating an invalid or unsupported command [16]. The receiver always closes the TCP connection even if the

UnRegisterSession command contains unexpected values, e.g., invalid session handle [16].

To test the UnRegisterSession commands, the Session Handle, Status, Sender Context, and Options fields are fuzzed. The Session Handle field is tested with a combination of inputs ranging from 0x00000000 to 0xFFFFFFFF. The Status field is fuzzed in a similar manner. The Sender Context field is tested from 0x0000000000000000 to 0xFFFFFFFFFFFFFFFF. The Options field is tested with values between 0x00000000 and 0xFFFFFFFF. Aside from the Session Handle test, the other fields are fuzzed using a valid Session Handle in the packet.

To determine if MicroLogix complies with the ENIP requirement that an UnRegisterSession command sent over UDP will be rejected with an error code of 0x01 "invalid or unsupported command" [16], a single properly-crafted UDP UnRegisterSession command is included in the ENIP test suite. Figures 12 and 13 display a sample TCP ENIP UnRegisterSession Request. Figures 14 and 15 show a UDP version of the command for exception testing purposes.

```
###[ ENIP TCP ]###
          Command   = UnRegister Session (0x0066)
          Length    = None
          Session_Handle= 0x0
          Status    = Success
          Sender_Context= 0
          Options   = 0
###[ Unregister Session ]###
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 12. An Example ENIP UnRegisterSession Request over TCP Packet

```
0000    00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010    00 40 00 01 00 00 40 06 f8 ed c0 a8 00 3e c0 a8
0020    00 3b f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030    20 00 05 73 00 00 66 00 00 00 00 00 00 00 00 00
0040    00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 13. Hexadecimal View of Example TCP ENIP UnRegisterSession Request Packet

```
###[ ENIP UDP ]###
        Command    = 0x66
        Length     = None
        Session_Handle= 0x0
        Status     = Success
        Sender_Context= 0
        Options    = 0
###[ Unregister Session ]###
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 14. An Example ENIP UnRegisterSession Request over UDP Packet.

```
0000    45 00 00 34 00 01 00 00 40 11 f8 ee c0 a8 00 3e
0010    c0 a8 00 3b f3 7a af 12 00 20 75 56 66 00 00 00
0020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030    00 00 00 00
```

Figure 15. Hexadecimal View of Example ENIP UnRegisterSession Request over UDP Packet

3. ENIP SendRRData Command

SendRRData sends encapsulated messages from an originator to a target. When encapsulating CIP, the SendRRData command transports unconnected messages [16]. To test the SendRRData command fields, the following fields are fuzzed using a TCP connection: Session Handle, Status, Sender Context, Options, Interface Handle, and Timeout fields. The Session Handle field is tested with a combination of inputs ranging from 0x00000000 to 0xFFFFFFFF. The Status field is fuzzed in a similar manner. The Sender Context field is tested with data ranging from 0x0000000000000000 to 0xFFFFFFFFFFFFFFFF. The Options field is tested between 0x00000000 and

0xFFFFFFFF. The Interface Handle is tested between 0x00000000 and 0xFFFFFFFF and the Timeout field is tested between 0 and 65535. For the Encapsulated Data field, a CIP Forward Open command is used. Figures 16 and 17 illustrate a sample SendRRData request containing an encapsulated CIP Forward Open Request [15].

```
###[ ENIP TCP ]###
           Command   = Send RR Data (0x006F)
           Length    = 62
           Session_Handle= 0x0
           Status    = Success
           Sender_Context= 0
           Options   = 0
###[ Send RR Data ]###
              Interface_Handle= 0
              Timeout   = 0
###[ ENIP_CommonPacketFormat ]###
                 Item_Count= None
                 \Items     \
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Null (0x0000)
                  |  Address_Length= None
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Unconnected Message (0x00B2)
                  |  Data_Length= 46
###[ Common_Industrial_Protocol ]###
                    Request_Response= Request
                    Common_Service= Connection_Manager_Forward_Open
                    Request_Path_Size= None
                    \Words     \
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Class ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Class     = Connection Manager Object
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Instance ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Eight_bit_Instance= 0x1
###[ CIP CM Forward Open Request ]###
                       Reserved  = 512
                       Priority  = Normal
                       Tick_Time = 128
                       Time_out_ticks= 155
                       O_T_Network_Connection_ID= 0x80000002
                       T_O_Network_Connection_ID= 0x80fe0001
                       Connection_Serial_Number= 0x2
                       Originator_Vendor_ID= Rockwell Software, Inc.
                       Originator_Serial_Number= 90180339
                       Connection_Timeout_Multiplier= 2
                       Reserved  = 0x200
                       O_T_RPI   = 1250000
                       O_T_Network_Connection_Parameters= 0x4312
                       T_O_RPI   = 1250000
                       T_O_Network_Connection_Parameters= 0x4312
                       Transport_Type_Trigger_Direction= Server
                       Transport_Type_Trigger_Production_Trigger=
Application Object
                       Transport_Type_Trigger_Transport_Class= Class 3
                       Connection_Path_Size= None
                       \Words     \
                        |###[ CIP Request Path ]###
                        |  Path_Segment_Type= Logical Segment
                        |  Logical_Segment_Type= Class ID
                        |  Logical_Segment_Format= 8-bit logical address
                        |  Class     = Message Router
                        |###[ CIP Request Path ]###
                        |  Path_Segment_Type= Logical Segment
                        |  Logical_Segment_Type= Instance ID
                        |  Logical_Segment_Format= 8-bit logical address
                        |  Eight_bit_Instance= 0x1
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 16. An Example ENIP SendRRData Request over TCP with an Encapsulated CIP Forward Open Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 7e 00 01 00 00 40 06 f8 af c0 a8 00 3e c0 a8
0020   00 3b f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 c2 cb 00 00 6f 00 3e 00 00 00 00 00 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 00 00 02 00 00 00 00 00 b2 00 2e 00 54 02
0060   20 06 24 01 07 9b 02 00 00 80 01 00 fe 80 02 00
0070   4d 00 f3 0a 60 05 02 00 02 00 d0 12 13 00 12 43
0080   d0 12 13 00 12 43 a3 02 20 02 24 01
```

Figure 17. Hexadecimal View of Example ENIP SendRRData Request over TCP Packet

4. ENIP SendUnitData Command

The SendUnitData command [16] sends encapsulated connected messages that rely on their own end-to-end transport. Both originators and targets can initiate the SendUnitData command over a TCP connection. SendUnitData and SendRRData use the same packet structure. The Session Handle, Status, Sender Context, Options, Interface Handle, and Timeout fields are tested in the same manner as for SendRRData. Figures 18 and 19 demonstrate a sample SendRRData packet structure.

```
###[ ENIP TCP ]###
          Command   = Send Unit Data (0x0070)
          Length    = 28
          Session_Handle= 0x0
          Status    = Success
          Sender_Context= 0
          Options   = 0
###[ Send Unit Data ]###
             Interface_Handle= 0
             Timeout    = 0
###[ ENIP_CommonPacketFormat ]###
                Item_Count= None
                \Items     \
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connection-Based (0x00A1)
                 |  Address_Length= 4
                 |  Connection_Identifier= 0x0
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connected Transport Packet (0x00B1)
                 |  Data_Length= 8
                 |  Sequence_Number= 0x2
###[ Common_Industrial_Protocol ]###
                   Request_Response= Request
                   Common_Service= Get_Attributes_All
                   Request_Path_Size= 2
                   \Words     \
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Class ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Class     = Identity Object
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Instance ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Eight_bit_Instance= 0x1
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 18. An Example ENIP SendUnitData Request over TCP with an Encapsulated CIP Get_Attribute_All Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 5c 00 01 00 00 40 06 f8 ef c0 a8 00 3e c0 a8
0020   00 1d f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 38 6f 00 00 70 00 1c 00 00 00 00 00 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 00 00 02 00 a1 00 04 00 00 00 00 00 b1 00
0060   08 00 02 00 01 02 20 01 24 01
```

Figure 19. Hexadecimal View of Example EtherNet/IP SendUnitData Request over TCP Packet

5. ENIP Reserved for Legacy Use Commands

In the CIP Networks Library: Volume 2 EtherNet/IP Adaptation of CIP specification [16], several commands are labeled as "Reserved for legacy use" (herein referred to as Legacy Use) with no explanation of their functionality or packet structure. The command codes for the Legacy Use commands are 0x0001, 0x0002, 0x0005, 0x0067-0x006E, and 0x0071-0x00C7. These commands are tested to determine if MicroLogix handles them as defined by the ENIP specification, i.e., commands that are not supported by a target device shall not break the session or TCP connection. This testing also aims to discover unknown functionality of the legacy commands. Testing is conducted over both TCP and UDP connections. Figures 20 and 21 show the structure of a sample ENIP Legacy Use command sent over TCP. Figures 22 and 23 show the structure of a sample Legacy Use command sent over UDP.

```
###[ ENIP TCP ]###
          Command   = Indicate Status (0x0072)
          Length    = None
          Session_Handle= 0x0
          Status    = Success
          Sender_Context= 0
          Options   = 0
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 20. An Example ENIP Legacy Use Request over TCP

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 40 00 01 00 00 40 06 f8 ed c0 a8 00 3e c0 a8
0020   00 3b f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 f9 72 00 00 72 00 00 00 00 00 00 00 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 21. Hexadecimal View of Example ENIP Legacy Use Request over TCP Packet

```
###[ ENIP UDP ]###
          Command    = 0x72
          Length     = None
          Session_Handle= 0x0
          Status     = Success
          Sender_Context= 0
          Options    = 0
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 22. An Example ENIP Legacy Use Request over UDP.

```
0000   45 00 00 34 00 01 00 00 40 11 f8 ee c0 a8 00 3e
0010   c0 a8 00 3b f3 7a af 12 00 20 69 56 72 00 00 00
0020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030   00 00 00 00
```

Figure 23. Hexadecimal View of Example ENIP Legacy Use Request over UDP Packet

6. ENIP Reserved for Future Use Commands

There are also designated ENIP commands that are labeled “Reserved for future use” (herein referred to as Future Use) in the CIP Networks Library: Volume 2 EtherNet/IP Adaptation of CIP specification [16]. The ranges of the Future Use commands are 0x0006-0x0062 and 0x00C8-0xFFFF. These commands are tested to determine if MicroLogix handles them as defined by the ENIP specification, i.e., commands that are not supported by a target device shall not break the session or TCP connection. Figures 24 and 25 show the structure of an ENIP Future Use command sent over TCP. Figures 26 and 27 show the structure of an ENIP Future Use command sent over UDP.

```
###[ ENIP TCP ]###
          Command   = Unknown Command (0x0006)
          Length    = None
          Session_Handle= 0x0
          Status    = Success
          Sender_Context= 0
          Options   = 0
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 24. An Example ENIP Future Use Request over TCP

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 40 00 01 00 00 40 06 f8 ed c0 a8 00 3e c0 a8
0020   00 3b f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 f9 72 00 00 06 00 00 00 00 00 00 00 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 25. Hexadecimal View of Example ENIP Future Use Request over TCP Packet

```
###[ ENIP UDP ]###
          Command   = 0x06
          Length    = None
          Session_Handle= 0x0
          Status    = Success
          Sender_Context= 0
          Options   = 0
```

Fields encapsulated at the ENIP layer are highlighted.

Figure 26. An Example UDP ENIP Future Use Request

```
0000   45 00 00 34 00 01 00 00 40 11 f8 ee c0 a8 00 3e
0010   c0 a8 00 3b f3 7a af 12 00 20 69 56 06 00 00 00
0020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030   00 00 00 00
```

Figure 27. Hexadecimal View of Example UPD ENIP Future Use Response Packet

C. CIP FUZZING TEST PLAN

Previous CIP fuzz testing is limited to the CIP NOP command [9]. This thesis expands the testing scope to include four additional CIP Common Services shown in the CIP Test Plan in Table 7. Command fields are tested in isolation. All tests use the ENIP command SendUnitData, which can only be used with TCP.

Table 7. CIP Test Plan

Test Number	CIP Command	Fuzzed Field	Protocol	Fuzzing Parameters
T30	Get_Attributes_All	Class	TCP	Class 0x00-0xFF, Attribute 0x01
T31	Get_Attributes_All	Instance	TCP	Class 0x01, Attribute 0x00-0xFF
T32	Get_Attribute_List	Class	TCP	Class 0x00-0xFF, Attribute_List 0x01, Instance 0x01
T33	Get_Attribute_List	Attribute_List	TCP	Class 0x01, Attribute_List 0x00-0xFF, Instance 0x01
T34	Get_Attribute_List	Instance	TCP	Class 0x01, Attribute_List 0x01, Instance 0x00-0xFF
T35	Get_Attribute_List	Attribute_count	TCP	Max Attribute_count Length (Increasing lengths of Attribute_count field)
T36	Get_Attribute_Single	Class	TCP	Class 0x00-0xFF, Attribute 0x01, Instance 0x00
T37	Get_Attribute_Single	Instance	TCP	Class 0x01, Attribute 0x01, Instance 0x00-0xFF
T38	Get_Attribute_Single	Attribute	TCP	Class 0x01, Attribute 0x00-0xFF, Instance 0x01
T39	Find_Next_Object_Instance	Class	TCP	Class 0x00-0xFF, Instance 0x00, Maximum Returned Values 0x00
T40	Find_Next_Object_Instance	Instance	TCP	Class 0x01, Instance 0x00-0xFF, Maximum Returned Values 0x00
T41	Find_Next_Object_Instance	Maximum Returned Values	TCP	Class 0x01, Instance 0x00, Maximum Returned Values 0x00

1. CIP Get_Attributes_All

The Get_Attributes_All command requests the contents of all instance or class attributes that the specified object supports [15]. Both Class and Attribute fields are individually fuzzed with values in the range of 0x00 to 0xFF. Figures 28 and 29 show the structure of a sample Get_Attributes_All command over TCP.

```
###[ ENIP TCP ]###
          Command   = Send Unit Data (0x0070)
          Length    = 28
          Session_Handle= 0x0
          Status    = Success
          Sender_Context= 0
          Options   = 0
###[ Send Unit Data ]###
             Interface_Handle= 0
             Timeout   = 0
###[ ENIP_CommonPacketFormat ]###
                Item_Count= None
                \Items     \
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connection-Based (0x00A1)
                 |  Address_Length= 4
                 |  Connection_Identifier= 0x0
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connected Transport Packet (0x00B1)
                 |  Data_Length= 8
                 |  Sequence_Number= 0x2
###[ Common_Industrial_Protocol ]###
                   Request_Response= Request
                   Common_Service= Get_Attributes_All
                   Request_Path_Size= 2
                   \Words     \
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Class ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Class     = Identity Object
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Instance ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Eight_bit_Instance= 0x1
```

Fields encapsulated at the CIP layer are highlighted.

Figure 28. An Example CIP Get_Attributes_All Request over TCP

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 5c 00 01 00 00 40 06 f8 ef c0 a8 00 3e c0 a8
0020   00 1d f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 38 6f 00 00 70 00 1c 00 00 00 00 00 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 00 00 02 00 a1 00 04 00 00 00 00 00 b1 00
0060   08 00 02 00 01 02 20 01 24 01
```

Figure 29. Hexadecimal View of Example CIP Get_Attributes_All Request over TCP Packet

2. CIP Get_Attribute_List

The Get_Attribute_List Command requests the selected attributes of an object class or instance [15]. The Get_Attribute_List is an optional service [2]. The Class, Attribute, and Instance fields are individually fuzzed with values in the range of 0x00-0xFF. The Attribute_count field is also tested by sending Get_Attribute_List requests with increasing values in the Attribute_count field up to 0xFFFF to determine the maximum number of attributes allowable. Figures 30 and 31 show the structure of a sample TCP Get_Attribute_List command.

```
###[ ENIP TCP ]###
           Command   = Send Unit Data (0x0070)
           Length    = 32
           Session_Handle= 0xf020100
           Status    = Success
           Sender_Context= 0
           Options   = 0
###[ Send Unit Data ]###
              Interface_Handle= 0
              Timeout   = 1
###[ ENIP_CommonPacketFormat ]###
                 Item_Count= 2
                 \Items     \
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connection-Based (0x00A1)
                  |  Address_Length= 4
                  |  Connection_Identifier= 0x9f9d0b6f
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connected Transport Packet (0x00B1)
                  |  Data_Length= 14
                  |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                    Request_Response= Request
                    Common_Service= Get_Attribute_List
                    Request_Path_Size= 2
                    \Words     \
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Class ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Class     = Identity
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Instance ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Eight_bit_Instance= 0x1
###[ CIP Get Attribute List Request ]###
                       Attribute_Count= 1
                       Attributes= ['1']
```

Fields encapsulated at the CIP layer are highlighted.

Figure 30. An Example CIP Get_Attribute_List Request over TCP

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 60 00 01 00 00 40 06 e4 92 0a 01 1e 01 0a 01
0020   64 02 f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 06 62 00 00 70 00 20 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   0e 00 01 00 03 02 20 01 24 01 01 00 01 00
```

Figure 31. Hexadecimal View of Example CIP Get_Attribute_List Request over TCP Packet

3. CIP Get_Attribute_Single

The Get_Attribute_Single command requests the contents of a specified attribute. This service is to be implemented for the Identity Object if any Class Attributes are implemented [15]. Class, Attribute, and Instance fields are fuzzed with values ranging from 0x00 to 0xFF. Figures 32 and 33 show the structure of a sample TCP Get_Attribute_Single command.

```
###[ ENIP TCP ]###
           Command   = Send Unit Data (0x0070)
           Length    = 30
           Session_Handle= 0xf020100
           Status    = Success
           Sender_Context= 0
           Options   = 0
###[ Send Unit Data ]###
              Interface_Handle= 0
              Timeout   = 1
###[ ENIP_CommonPacketFormat ]###
                 Item_Count= 2
                 \Items     \
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connection-Based (0x00A1)
                  |  Address_Length= 4
                  |  Connection_Identifier= 0x9f9d0b6f
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connected Transport Packet (0x00B1)
                  |  Data_Length= 10
                  |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                    Request_Response= Request
                    Common_Service= Get_Attribute_Single
                    Request_Path_Size= 2
                    \Words     \
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Class ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Class     = Identity Object
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Instance ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Eight_bit_Instance= 0x1
###[ CIP Get Attribute Single Request ]###
                       Attribute_Identifier= 1
```

Fields encapsulated at the CIP layer are highlighted.

Figure 32. An Example CIP Get_Attribute_Single Request over TCP

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 5d 00 01 00 00 40 06 e4 95 0a 01 1e 01 0a 01
0020   64 02 f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 05 5a 00 00 70 00 1d 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   09 00 01 00 0e 02 20 01 24 01 01 00
```

Figure 33. Hexadecimal View of Example CIP Get_Attribute_Single Request over TCP Packet

4. CIP Find_Next_Object_Instance

The Find_Next_Object_Instance command requests a list of Instance IDs associated with existing Object Instances that are accessible from the CIP subnet at the time the request is made [15]. The request command specifies the number of requested Instances, but the number of returned Instances can be less. If the Instance ID in the request is zero, the Instance ID that is numerically lowest in the Class is returned [15]. If the Instance ID in the request is less than the highest Instance ID in the Class, successful responses return the next Instance ID that is numerically higher than the Instance ID specified in the request [2]. If the Instance ID in the request is greater than or equal to the highest Instance ID in the Class, the value 0 is returned [15]. This service is only available at the Class level [15]. Testing is conducted on the Class, Instance, and Maximum Returned Values fields of this command with inputs ranging from 0x00 to 0xFF. Figures 34 and 35 show the structure of a sample CIP Find_Next_Object_Instance command over TCP.

```
###[ ENIP TCP ]###
          Command   = Send RR Data (0x006F)
          Length    = None
          Session_Handle= 0xf020100
          Status    = Success
          Sender_Context= 0
          Options   = 0
###[ Send RR Data ]###
             Interface_Handle= 0
             Timeout   = 0
###[ ENIP_CommonPacketFormat ]###
                Item_Count= None
                \Items     \
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Null (0x0000)
                 |  Address_Length= 4
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Unconnected Message (0x00B2)
                 |  Data_Length= 7
###[ Common_Industrial_Protocol ]###
                  Request_Response= Request
                  Common_Service= Find_Next_Object_Instance
                  Request_Path_Size= 2
                  \Words     \
                   |###[ CIP Request Path ]###
                   |  Path_Segment_Type= Logical Segment
                   |  Logical_Segment_Type= Class ID
                   |  Logical_Segment_Format= 8-bit logical address
                   |  Class     = Identity Object
                   |###[ CIP Request Path ]###
                   |  Path_Segment_Type= Logical Segment
                   |  Logical_Segment_Type= Instance ID
                   |  Logical_Segment_Format= 8-bit logical address
                   |  Eight_bit_Instance= 0x1
###[ CIP_Find_Next_Object_Instance_Request ]###
                    Max_Returned_Vals= 1
```

Fields encapsulated at the CIP layer are highlighted.

Figure 34. An Example CIP Find_Next_Object_Instance Request over TCP

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 57 00 01 00 00 40 06 f8 d6 c0 a8 00 3e c0 a8
0020   00 3b f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 ce 46 00 00 6f 00 17 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 00 00 02 00 00 00 04 00 b2 00 07 00 11 02
0060   20 01 24 01 01
```

Figure 35. Hexadecimal View of Example CIP Find_Next_Object_Instance Request over TCP Packet

D. PCCC FUZZING TEST PLAN

Previous MicroLogix PCCC fuzz testing is limited to the Execute PCCC command Protected Typed Logical Read with Three Address Fields [17]. This thesis expands the MicroLogix testing to fuzz PCCC commands not tested by Tacliad [9] for vulnerabilities. Command fields are tested in isolation on the MicroLogix PLC in order to provide a methodical evaluation of each command's potential vulnerabilities.

Additionally, to determine if a recently discovered MicroLogix PCCC vulnerability affects the ControlLogix, the Protected Logical Read with Three Address Fields is tested on the ControlLogix with a MicroLogix fault-causing combination of field inputs. Table 8 summarizes the PCCC test plan.

Table 8. PCCC Testing Plan

Test Number	PCCC Command	Fuzzed Field	Protocol	Fuzzing Parameters
MicroLogix Tests				
T42	Echo	Data: 0 bytes	TCP	0 Attached bytes
T43	Echo	Data: Max Length	TCP	Increasing number of attached bytes
T44	Echo	Data: 8 bytes	TCP	8 Attached random bytes
T45	Echo	Data: 9 bytes	TCP	9 Attached random bytes
T46	Echo	Data: 10 bytes	TCP	10 Attached random bytes
T47	Echo	Data: 40 bytes	TCP	40 Attached random bytes
T48	Echo	Data: 243 bytes	TCP	243 Attached random bytes
T49	Echo	Data: Maximum bytes returned by module with no errors	TCP	Maximum random bytes returned by module with no error
T50	Echo	Data: 248 bytes	TCP	248 Attached random bytes
T51	Echo	Data: 256 bytes	TCP	256 Attached random bytes
T52	Protected Typed File Read	Size	TCP	Size (0x00-0xFF)

Test Number	PCCC Command	Fuzzed Field	Protocol	Fuzzing Parameters
T53	Protected Typed File Read	Tag	TCP	Tag (0x0000-0xFFFF)
T54	Protected Typed File Read	Offset	TCP	Offset (0x0000-0xFFFF)
T55	Protected Typed File Read	File Type	TCP	File Type (0x00-0xFF)
T56	Protected Typed File Write	Size	TCP	Size (0x00-0xFF)
T57	Protected Typed File Write	Tag	TCP	Tag (0x0000-0xFFFF)
T58	Protected Typed File Write	Offset	TCP	Offset (0x0000-0xFFFF)
T59	Protected Typed File Write	File Type	TCP	File Type (0x00-0xFF)
T60	Protected Typed File Write	Data	TCP	Data (0x00-0xFF)
T61	Protected Typed Logical Write with Three Address Fields	Byte Size	TCP	Byte Size (0x00-0xFF)
T62	Protected Typed Logical Write with Three Address Fields	File No.	TCP	File No. (0x00-0xFF)
T63	Protected Typed Logical Write with Three Address Fields	File Type	TCP	File Type (0x00-0xFF)
T64	Protected Typed Logical Write with Three Address Fields	Element No.	TCP	Element No. (0x00-0xFF and 0xFF0000-0xFFFFFF)
T65	Protected Typed Logical Write with Three Address Fields	Sub-Element No.	TCP	Sub-Element No. (0x00-0xFF and 0xFF0000-0xFFFFFF)
T66	Unprotected Read	Address	TCP	Address (0x0000-0xFFFF)
T67	Unprotected Read	Size	TCP	Size (0x00-0xFF)
T68	Diagnostic Status-Functionality Test	N/A	TCP	Properly formatted command
T69	Read Diagnostic Counters	Address	TCP	Address (0x0000-0xFFFF)

Test Number	PCCC Command	Fuzzed Field	Protocol	Fuzzing Parameters
T70	Read Diagnostic Counters	Size	TCP	Size (0x00-0xFF)
T71	Restart-Functionality Test	N/A	TCP	Properly formatted command
T72	Download Completed-Functionality Test	N/A	TCP	Properly formatted command
ControlLogix Tests				
T73	Protected Typed Logical Read with Three Address Fields	File No., File Type	TCP	File No. (0x2-0x8), File Type (0x47-0x48)

1. PCCC Echo Command

The Echo command enables a user to check the integrity of a communication link. The receiving module replies to a request with the same data in the original transmission. According Allen-Bradley's DF1 Protocol and Command Set specification [17], this command is compatible with the MicroLogix 1000, a member of the MicroLogix 1100 family of products, and should transmit a maximum of 243 bytes of data. In order to test the maximum data allowable in an Echo command, the fuzzing device sends commands with an increasing number of repeating bytes, starting from 0 to the maximum size that the receiving module will reply with no errors, while monitoring SUT responses. Echo commands are tested with random bytes using the following lengths: 0, 8, 9, 10, 243, 248, 256, and the observed maximum size returned with no errors. Figures 36 and 37 illustrate the structure of the PCCC Echo command.

```
###[ ENIP TCP ]###
          Command   = Send Unit Data (0x0070)
          Length    = 42
          Session_Handle= 0xf020100
          Status    = Success
          Sender_Context= 0
          Options   = 0
###[ Send Unit Data ]###
             Interface_Handle= 0
             Timeout    = 1
###[ ENIP_CommonPacketFormat ]###
                Item_Count= 2
                \Items     \
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connection-Based (0x00A1)
                 |  Address_Length= 4
                 |  Connection_Identifier= 0x9f9d0b6f
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connected Transport Packet (0x00B1)
                 |  Data_Length= 23
                 |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                   Request_Response= Request
                   Common_Service= Execute_PCCC_Service
                   Request_Path_Size= 2
                   \Words     \
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Class ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Class     = 0x67
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Instance ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                      Length_of_Requestor_ID= 7
                      CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                      CIP_Serial_Number= 90180339
                      CMD       = 0x06
                      Status    = 0x0
                      Transaction_Word= 1
                      Function  = Echo
###[ Padding ]###
                         load      = '\x01\x02'
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 36. An Example PCCC Echo Request with Two Data Bytes over TCP

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 6a 00 01 00 00 40 06 e4 88 0a 01 1e 01 0a 01
0020   64 02 f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 95 3d 00 00 70 00 2a 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   17 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 06 00 01 00 00 01 02
```

Figure 37. Hexadecimal View of Example PCCC Echo Request over TCP Packet.

2. PCCC Protected Typed File Read

The Protected Typed File Read command reads data from an open file [17]. Four fields are fuzz tested: Size, Tag, Offset, and File Type. The one-byte fields, Size and File Type, are tested with random inputs from 0x00 to 0xFF. The two-byte fields, Tag and Offset, are tested with random inputs from 0x0000 to 0xFFFF. The SUT is expected to provide successful read responses. Figures 38 and 39 illustrate the structure of an example packet.

```
###[ ENIP TCP ]###
            Command   = Send Unit Data (0x0070)
            Length    = None
            Session_Handle= 0xf020100
            Status    = Success
            Sender_Context= 0
            Options   = 0
###[ Send Unit Data ]###
               Interface_Handle= 0
               Timeout   = 1
###[ ENIP_CommonPacketFormat ]###
                  Item_Count= None
                  \Items     \
                   |###[ Common Packet Format Item ]###
                   |  Address_Data_Item= Connection-Based (0x00A1)
                   |  Address_Length= 4
                   |  Connection_Identifier= 0x9f9d0b6f
                   |###[ Common Packet Format Item ]###
                   |  Address_Data_Item= Connected Transport Packet (0x00B1)
                   |  Data_Length= 26
                   |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                     Request_Response= Request
                     Common_Service= Execute_PCCC_Service
                     Request_Path_Size= None
                     \Words     \
                      |###[ CIP Request Path ]###
                      |  Path_Segment_Type= Logical Segment
                      |  Logical_Segment_Type= Class ID
                      |  Logical_Segment_Format= 8-bit logical address
                      |  Class     = 0x67
                      |###[ CIP Request Path ]###
                      |  Path_Segment_Type= Logical Segment
                      |  Logical_Segment_Type= Instance ID
                      |  Logical_Segment_Format= 8-bit logical address
                      |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                        Length_of_Requestor_ID= 7
                        CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                        CIP_Serial_Number= 90180339
                        CMD       = 0x0F
                        Status    = 0x0
                        Transaction_Word= 2
                        Function  = Protected Typed File Read
                        Size      = 0xc0
                        Tag       = 33765
                        Offset    = 36443
                        File_Type = 0xc0
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 38. An Example PCCC Protected Typed File Read Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 6e 00 01 00 00 40 06 f8 bf c0 a8 00 3e c0 a8
0020   00 3b f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 cf c3 00 00 70 00 2e 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   1a 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 0f 00 02 00 a7 c0 e5 83 5b 8e c0
```

Figure 39. Hexadecimal View of Example PCCC Protected Typed File Read Packet

3. PCCC Protected Typed File Write

The Protected Typed File Write command writes data to an open file in the PLC [17]. Testing is conducted on five fields: Size, Tag, Offset, File Type, and Data. The one-byte fields, Size and File Type, are tested with random inputs from 0x00 to 0xFF. The two-byte fields, Tag and Offset, are tested with random inputs from 0x0000 to 0xFFFF. The data field is tested with a two-byte size with random inputs from 0x0000 to 0xFFFF. The SUT is expected to provide successful write responses. Figures 40 and 41 illustrate the structure of an example packet.

```
###[ ENIP TCP ]###
           Command   = Send Unit Data (0x0070)
           Length    = None
           Session_Handle= 0xf020100
           Status    = Success
           Sender_Context= 0
           Options   = 0
###[ Send Unit Data ]###
              Interface_Handle= 0
              Timeout   = 1
###[ ENIP_CommonPacketFormat ]###
                 Item_Count= None
                 \Items     \
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connection-Based (0x00A1)
                  |  Address_Length= 4
                  |  Connection_Identifier= 0x9f9d0b6f
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connected Transport Packet (0x00B1)
                  |  Data_Length= 28
                  |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                    Request_Response= Request
                    Common_Service= Execute_PCCC_Service
                    Request_Path_Size= None
                    \Words      \
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Class ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Class     = 0x67
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Instance ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                       Length_of_Requestor_ID= 7
                       CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                       CIP_Serial_Number= 90180339
                       CMD       = 0x0F
                       Status    = 0x0
                       Transaction_Word= 2
                       Function  = Protected_Typed_File_Write
                       Size      = 0x2
                       Tag       = 4176
                       Offset    = 0
                       File_Type = 0x5a
                       Data      = 65535
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 40. An Example PCCC Protected Typed File Write Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 70 00 01 00 00 40 06 f8 bd c0 a8 00 3e c0 a8
0020   00 3b f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 8c 11 00 00 70 00 30 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   1c 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 0f 00 02 00 af 02 50 10 00 00 5a ff ff
```

Figure 41. Hexadecimal View of Example PCCC Protected Typed File Write Packet

4. PCCC Protected Typed Logical Write with Three Address Fields

The Protected Logical Write with Three Address Fields command writes data to a logical address in the PLC's processor [17]. The specification [17] is unclear whether the MicroLogix family of PLCs supports this command. Specifically, while the table that summarizes the PCCC commands and compatible processors indicates MicroLogix supports the command, the detailed description of this particular command omits MicroLogix as a supporting platform. Based on previous testing of the Protected Logical Read with Three Address Fields command [9], the MicroLogix is assumed to support the command. Testing is conducted on the fields Byte Size, File Number, and File Type with inputs ranging from 0x00 to 0xFF. Element Number, and Sub-element Number are one-byte fields that can expand to three bytes when the first byte is set to 0xFF. In this case, the second and third bytes identify the expanded sub-element [17]. For this reason, these fields are tested in the one-byte configuration with inputs ranging from 0x00 to 0xFF and in the three-byte configuration with inputs ranging from 0xFF0000 to 0xFFFFFF. The Data field is not fuzzed in an effort to avoid overwriting memory space with unknown functionality. Figures 42 and 43 show the structure of an example packet.

```
###[ ENIP TCP ]###
          Command   = Send Unit Data (0x0070)
          Length    = 45
          Session_Handle= 0xf020100
          Status    = Success
          Sender_Context= 0
          Options   = 0
###[ Send Unit Data ]###
             Interface_Handle= 0
             Timeout   = 1
###[ ENIP_CommonPacketFormat ]###
                Item_Count= None
                \Items     \
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connection-Based (0x00A1)
                 |  Address_Length= 4
                 |  Connection_Identifier= 0x9f9d0b6f
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connected Transport Packet (0x00B1)
                 |  Data_Length= 25
                 |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                   Request_Response= Request
                   Common_Service= Execute_PCCC_Service
                   Request_Path_Size= None
                   \Words     \
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Class ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Class     = 0x67
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Instance ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                      Length_of_Requestor_ID= 7
                      CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                      CIP_Serial_Number= 90180339
                      CMD       = 0x0F
                      Status    = 0x0
                      Transaction_Word= 2
                      Function  = Protected_Typed_Logical_Write_Three_Address_Fields
                      Byte_Size = 0x1
                      File_No   = 0x1
                      File_Type = 0x1
                      Element_No= 0x0
                      Sub_Element_No= 0x0
                      Data      = None
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 42. An Example PCCC Protected Typed Logical Write with Three Address Fields Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 6e 00 01 00 00 40 06 f8 bf c0 a8 00 3e c0 a8
0020   00 3b f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 a1 c1 00 00 70 00 2e 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   19 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 0f 00 02 00 aa 01 01 01 00 00
```

Figure 43. Hexadecimal View of Example PCCC Protected Typed Logical Write with Three Address Fields Request over TCP Packet

5. PCCC Unprotected Read

The Unprotected Read command requests data from a common interface file on the PLC [17] Fuzz testing is conducted on two fields: Address and Size. The two-byte Address field is fuzzed with random numbers between 0x0000 to 0xFFFF. The one-byte Size field is fuzzed with inputs between 0x00 to 0xFF. The expected result of the MicroLogix testing is a successful read response from the SUT. Figures 44 and 45 show the structure of an example packet.

```
###[ ENIP TCP ]###
           Command    = Send Unit Data (0x0070)
           Length     = None
           Session_Handle= 0xf020100
           Status     = Success
           Sender_Context= 0
           Options    = 0
###[ Send Unit Data ]###
              Interface_Handle= 0
              Timeout    = 1
###[ ENIP_CommonPacketFormat ]###
                 Item_Count= None
                 \Items     \
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connection-Based (0x00A1)
                  |  Address_Length= 4
                  |  Connection_Identifier= 0x9f9d0b6f
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connected Transport Packet (0x00B1)
                  |  Data_Length= 22
                  |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                    Request_Response= Request
                    Common_Service= Execute_PCCC_Service
                    Request_Path_Size= None
                    \Words     \
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Class ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Class     = 0x67
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Instance ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                       Length_of_Requestor_ID= 7
                       CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                       CIP_Serial_Number= 90180339
                       CMD        = Unprotected_Read
                       Status     = 0x0
                       Transaction_Word= 2
                       Address    = 0
                       Size       = 0x1
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 44. An Example PCCC Unprotected Read Request

Figure 45. Hexadecimal View of Example PCCC Unprotected Read Packet

6. PCCC Diagnostic Status

The Diagnostic Status command requests up to 244 bytes of status information from an interface module. Per the specification [17], the MicroLogix 1000 implementation of the command provides information including firmware, processor mode, and processor random access memory (RAM) size for the interface (24 bytes [17]). Documentation specific to the MicroLogix 1100 implementation of the command is not available. This command has no input parameter to fuzz, and thus, it is only functionally tested to determine MicroLogix 1100-specific responses. Figures 46 and 47 show the structure of an example packet.

```
###[ ENIP TCP ]###
           Command   = Send Unit Data (0x0070)
           Length    = 40
           Session_Handle= 0xf020100
           Status    = Success
           Sender_Context= 0
           Options   = 0
###[ Send Unit Data ]###
              Interface_Handle= 0
              Timeout   = 1
###[ ENIP_CommonPacketFormat ]###
                 Item_Count= 2
                 \Items     \
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connection-Based (0x00A1)
                  |  Address_Length= 4
                  |  Connection_Identifier= 0x9f9d0b6f
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connected Transport Packet (0x00B1)
                  |  Data_Length= 20
                  |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                    Request_Response= Request
                    Common_Service= Execute_PCCC_Service
                    Request_Path_Size= 2
                    \Words     \
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Class ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Class     = 0x67
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Instance ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                       Length_of_Requestor_ID= 7
                       CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                       CIP_Serial_Number= 90180339
                       CMD       = 0x06
                       Status    = 0x0
                       Transaction_Word= 2
                       Function  = Diagnostic_Status
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 46. An Example PCCC Diagnostic Status over TCP Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 68 00 01 00 00 40 06 e4 8a 0a 01 1e 01 0a 01
0020   64 02 f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 9b 3d 00 00 70 00 28 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   14 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 06 00 02 00 03
```

Figure 47. Hexadecimal View of Example PCCC Diagnostic Status Request over TCP Packet

7. PCCC Read Diagnostic Counters

Per the specification [17], the MicroLogix 1000 implementation of the command is used to read a module’s diagnostic timers and counters by requesting up to 244 bytes of data from the programmable read-only memory (PROM) or RAM of an interface module [17]. The specification does not provide any information specific to the MicroLogix 1100 implementation of the command. This command has two input parameters: Address and Size. The Address field is fuzzed between 0x0000 and 0xFFFF with a Size field set to 0x01. The Size field is fuzzed between 0x00 and 0xFF with the Address field set to 0x0000. Figures 48 and 49 illustrate the structure of an example PCCC Read Diagnostic Counters packet.

```
###[ ENIP TCP ]###
          Command    = Send Unit Data (0x0070)
          Length     = 43
          Session_Handle= 0xf020100
          Status     = Success
          Sender_Context= 0
          Options    = 0
###[ Send Unit Data ]###
             Interface_Handle= 0
             Timeout    = 1
###[ ENIP_CommonPacketFormat ]###
                Item_Count= 2
                \Items      \
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connection-Based (0x00A1)
                 |  Address_Length= 4
                 |  Connection_Identifier= 0x9f9d0b6f
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connected Transport Packet (0x00B1)
                 |  Data_Length= 23
                 |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                   Request_Response= Request
                   Common_Service= Execute_PCCC_Service
                   Request_Path_Size= 2
                   \Words      \
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Class ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Class      = 0x67
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Instance ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                      Length_of_Requestor_ID= 7
                      CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                      CIP_Serial_Number= 90180339
                      CMD        = 0x06
                      Status     = 0x0
                      Transaction_Word= 2
                      Function   = Read Diagnostic Counters
                      Address    = 0
                      Size       = 0x1
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 48. An Example PCCC Read Diagnostic Counters Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 6b 00 01 00 00 40 06 e4 87 0a 01 1e 01 0a 01
0020   64 02 f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 94 3c 00 00 70 00 2b 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   17 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 06 00 02 00 01 00 00 01
```

Figure 49. Hexadecimal View of Example PCCC Read Diagnostic Counters Request Packet

8. PCCC Restart

The PLC Restart command is intended solely for the PLC-3 and is not compatible with the MicroLogix family per the specification [17]. The command terminates any upload or download, revokes upload/download privileges, and initializes a PLC-3 restart. This command is tested with a properly formatted command in order to determine MicroLogix 1100 functionality. Figures 50 and 51 illustrate the structure of an example PCCC Restart request packet.

```
###[ ENIP TCP ]###
          Command    = Send Unit Data (0x0070)
          Length     = 40
          Session_Handle= 0xf020100
          Status     = Success
          Sender_Context= 0
          Options    = 0
###[ Send Unit Data ]###
             Interface_Handle= 0
             Timeout    = 1
###[ ENIP_CommonPacketFormat ]###
                Item_Count= 2
                \Items     \
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connection-Based (0x00A1)
                 |  Address_Length= 4
                 |  Connection_Identifier= 0x9f9d0b6f
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connected Transport Packet (0x00B1)
                 |  Data_Length= 20
                 |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                   Request_Response= Request
                   Common_Service= Execute_PCCC_Service
                   Request_Path_Size= 2
                   \Words     \
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Class ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Class     = 0x67
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Instance ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                      Length_of_Requestor_ID= 7
                      CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                      CIP_Serial_Number= 90180339
                      CMD       = 0x0F
                      Status    = 0x0
                      Transaction_Word= 2
                      Function  = Restart
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 50. An Example PCCC Restart Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 68 00 01 00 00 40 06 e4 8a 0a 01 1e 01 0a 01
0020   64 02 f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 9b 2d 00 00 70 00 28 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   14 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 0f 00 02 00 0a
```

Figure 51. Hexadecimal View of Example PCCC Restart Request Packet

9. PCCC Download Completed

The Download Completed command returns a processor to its previous mode upon completion of a complete system download [17]. This command is not intended for the MicroLogix PLC family. Functionality testing is conducted to observe MicroLogix 1100 responses to an illegal command. Figures 52 and 53 illustrate the structure of an example PCCC Download Completed packet.

```
###[ ENIP TCP ]###
           Command   = Send Unit Data (0x0070)
           Length    = 40
           Session_Handle= 0xf020100
           Status    = Success
           Sender_Context= 0
           Options   = 0
###[ Send Unit Data ]###
              Interface_Handle= 0
              Timeout   = 1
###[ ENIP_CommonPacketFormat ]###
                 Item_Count= 2
                 \Items     \
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connection-Based (0x00A1)
                  |  Address_Length= 4
                  |  Connection_Identifier= 0x9f9d0b6f
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connected Transport Packet (0x00B1)
                  |  Data_Length= 20
                  |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                    Request_Response= Request
                    Common_Service= Execute_PCCC_Service
                    Request_Path_Size= 2
                    \Words     \
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Class ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Class     = 0x67
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Instance ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                       Length_of_Requestor_ID= 7
                       CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                       CIP_Serial_Number= 90180339
                       CMD        = 0x0F
                       Status     = 0x0
                       Transaction_Word= 2
                       Function   = Download Completed
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 52. An Example PCCC Download Completed Request

0000	00	1d	9c	ca	cb	1b	90	e2	ba	18	fc	2e	08	00	45	00
0010	00	68	00	01	00	00	40	06	e4	8a	0a	01	1e	01	0a	01
0020	64	02	f3	7a	af	12	00	00	00	00	00	00	00	00	50	02
0030	20	00	9a	e5	00	00	70	00	28	00	00	01	02	0f	00	00
0040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0050	00	00	01	00	02	00	a1	00	04	00	6f	0b	9d	9f	b1	00
0060	14	00	01	00	4b	02	20	67	24	01	07	4d	00	f3	0a	60
0070	05	0f	00	02	00	52										

Figure 53. Hexadecimal View of Example PCCC Download Completed Request Packet

10. PCCC Protected Logical Read with Three Address Fields Command on ControlLogix

The Protected Logical Read with Three Address Fields is tested on the ControlLogix PLC to address this thesis' secondary research question: whether vulnerabilities discovered on earlier model AB/RA PLCs affect more advanced and modern AB/RA PLCs. Previous ENIP Fuzz testing led to the discovery of a vulnerability in MicroLogix's implementation of the command. When any combination of a File Number 0x2 to 0x8 and File Type of 0x47 or 0x48 is present in the command, the MicroLogix 1100 experiences a Major Error (0x8) and enters a fault state [9].

To test the ControlLogix, the fuzzer sends Protected Logical Read with Three Address Field commands with a File Number between 0x2 and 0x8 and File Type of 0x47 or 0x48 to determine if the ControlLogix is susceptible to the same vulnerability affecting MicroLogix PLCs. Figures 54 and 55 illustrate the structure of an example PCCC Download Completed packet.

```
###[ ENIP TCP ]###
           Command    = Send Unit Data (0x0070)
           Length     = 45
           Session_Handle= 0xf020100
           Status     = Success
           Sender_Context= 0
           Options    = 0
###[ Send Unit Data ]###
              Interface_Handle= 0
              Timeout    = 1
###[ ENIP_CommonPacketFormat ]###
                 Item_Count= 2
                 \Items     \
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connection-Based (0x00A1)
                  |  Address_Length= 4
                  |  Connection_Identifier= 0x9f9d0b6f
                  |###[ Common Packet Format Item ]###
                  |  Address_Data_Item= Connected Transport Packet (0x00B1)
                  |  Data_Length= 25
                  |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                    Request_Response= Request
                    Common_Service= Execute_PCCC_Service
                    Request_Path_Size= 2
                    \Words     \
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Class ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Class      = 0x67
                     |###[ CIP Request Path ]###
                     |  Path_Segment_Type= Logical Segment
                     |  Logical_Segment_Type= Instance ID
                     |  Logical_Segment_Format= 8-bit logical address
                     |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                       Length_of_Requestor_ID= 7
                       CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                       CIP_Serial_Number= 90180339
                       CMD        = 0x0F
                       Status     = 0x0
                       Transaction_Word= 1
                       Function   = Protected_Typed_Logical_Read_Three_Address_Fields
                       Byte_Size  = 0x1
                       File_No    = 0x1
                       File_Type  = 0x1
                       Element_No= 0x1
                       Sub_Element_No= 0x0
```

Fields encapsulated at the PCCC layer are highlighted.

Figure 54. An Example PCCC Protected Logical Read with Three Address Fields Request

```
0000   00 1d 9c ca cb 1b 90 e2 ba 18 fc 2e 08 00 45 00
0010   00 6d 00 01 00 00 40 06 e4 85 0a 01 1e 01 0a 01
0020   64 02 f3 7a af 12 00 00 00 00 00 00 00 00 50 02
0030   20 00 8e 8f 00 00 70 00 2d 00 00 01 02 0f 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 01 00 02 00 a1 00 04 00 6f 0b 9d 9f b1 00
0060   19 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 0f 00 01 00 a2 01 01 01 01 00
```

Figure 55. Hexadecimal View of Example PCCC Protected Logical Read with Three Address Fields Request Packet

THIS PAGE INTENTIONALLY LEFT BLANK

V. TEST ANALYSIS

This chapter presents the results of the fuzzed commands and a detailed analysis for each test case examined. Results are summarized first and subsequently expanded upon in the individual command result sections. Wireshark captures of SUT responses are included in Appendixes A through C.

A. ENIP TEST RESULTS

The ENIP tests do not cause any faults or disruption of service to the MicroLogix SUT. However, the testing does reveal several instances where the MicroLogix implementation of ENIP deviates from the specification [16]. Table 9 summarizes both expected and observed responses to the test cases.

Table 9. ENIP Fuzz Testing Results

Test Number	ENIP Command	Fuzzed Field	Protocol	Expected Fuzzed Response	Actual Fuzzed Response
T1	List Services/Identity/Interfaces	Session Handle	TCP	Session Handle repeated in response (ignored by target)	Session Handle repeated in response (ignored by target)
T2	List Services/Identity/Interfaces	Session Handle	UDP	Session Handle repeated in response	Session Handle repeated in response
T3	List Services/Identity/Interfaces	Status	TCP	TCP ACK	TCP ACK
T4	List Services/Identity/Interfaces	Status	UDP	No response	No response
T5	List Services/Identity/Interfaces	Sender Context	TCP	Sender Context repeated in response	Sender Context repeated in response
T6	List Services/Identity/Interfaces	Sender Context	UDP	Sender Context repeated in response	Sender Context repeated in response

Test Number	ENIP Command	Fuzzed Field	Protocol	Expected Fuzzed Response	Actual Fuzzed Response
T7	List Services/Identity/Interfaces	Options	TCP	Packet discarded	Deviation: See ListServices Results section
T8	List Services/Identity/Interfaces	Options	UDP	Packet discarded	Deviation: See ListServices Results section
T9	UnRegisterSession	Session Handle	TCP	Error 0x03 TCP close	Deviation: Error 0x03 No TCP close
T10	UnRegisterSession	Status	TCP	Error 0x03 TCP close	Deviation: Error 0x03 No TCP close
T11	UnRegisterSession	Sender Context	TCP	TCP close	TCP close
T12	UnRegisterSession	Options	TCP	Error 0x03 TCP close	Deviation: Error 0x03, no TCP close
T13	UnRegisterSession UDP Functionality	N/A	UDP	Error 0x01	Error 0x01
T14	SendRRData	Session Handle	TCP	Error 0x03	Deviation: See SendRRData Results Section
T15	SendRRData	Status	TCP	TCP ACK	Deviation: See SendRRData Results Section
T16	SendRRData	Sender Context	TCP	Successful Response with Sender Context returned	Successful Response with Sender Context returned
T17	SendRRData	Options	TCP	TCP ACK	Deviation: See SendRRData Results Section
T18	SendRRData	Interface Handle	TCP	Error 0x03	Error 0x03

Test Number	ENIP Command	Fuzzed Field	Protocol	Expected Fuzzed Response	Actual Fuzzed Response
T19	SendRRData	TimeOut	TCP	Error 0x03	Deviation: See SendRRData Results Section
T20	SendUnitData	Session Handle	TCP	Error 0x03	Deviation: See SendUnitData Results Section
T21	SendUnitData	Status	TCP	TCP ACK	Deviation: See SendUnitData Results Section
T22	SendUnitData	Sender Context	TCP	Successful Response with Sender Context returned	Successful Response with Sender Context returned
T23	SendUnitData	Options	TCP	TCP ACK	Deviation: See SendUnitData Results Section
T24	SendUnitData	Interface Handle	TCP	Error 0x03	Error 0x03
T25	SendUnitData	TimeOut	TCP	Error 0x03	Deviation: See SendUnitData Results Section
T26	Reserved for Legacy	Command Field	TCP	Error 0x03 or success	Error 0x03 or success
T27	Reserved for Legacy	Command Field	UDP	Error 0x01 or success	Error 0x01 or success
T28	Reserved for Future Use	Command Field	TCP	Error 0x03	Error 0x03
T29	Reserved for Future Use	Command Field	UDP	Error 0x01	Error 0x01

1. ENIP ListServices Results

a. T1 and T2 Test Cases

The SUT responds as expected to the ListServices commands (TCP and UDP) with non-zero values in the Session Handle field, i.e., by ignoring the specified session handle and returning the same session handle for the established session in the response.

b. T3 and T4 Test Cases

When non-zero values are sent in the Status field, the SUT responds as expected, i.e., by returning a TCP FIN, ACK if the command is sent over TCP and dropping the packet if the command is sent over UPD.

c. T5 and T6 Test Cases

The SUT responds predictably to the ListServices commands with fuzzed Sender Context fields over both TCP and UDP, i.e., by returning the same Sender Context value in the response.

d. T7 and T8 Test Cases

Per the ENIP specification, receivers must discard any ENIP ListServices packets with non-zero values in the Options field [16]. For both TCP and UDP, the SUT does not discard the ListServices command with a non-zero value in the Options field, but sends a ENIP response with a 0x03 “Incorrect data” [16] error code.

2. ENIP UnRegisterSession Results

a. T9 Test Case

For the Session Handle field, the MicroLogix implementation of the UnRegisterSesssion command returns a 0x03 “Incorrect data” [16] response and does not terminate the TCP connect as expected. This is a deviation from the specification that dictates a “receiver shall not reject the UnRegisterSession due to unexpected values in the encapsulation header,” including invalid Session Handles and non-zero Status inputs [16].

b. T10 Test Case

When the UnRegisterSession command is fuzzed with invalid Status codes and valid Session Handles, the MicroLogix returns a 0x03 “Incorrect data” [16] error and the TCP connection is not terminated. This is a deviation from the specification observed in T9 with invalid Session Handle inputs.

c. T11 Test Case

When fuzzing the Sender Context field, the MicroLogix implementation of the UnRegisterSesssion command returns expected responses and terminates the TCP connection. This complies with the ENIP requirement that receivers do not reject UnRegisterSession commands with unexpected values in the encapsulation header but close the underlying TCP connection instead [16].

d. T12 Test Case

When the Options field is set to a non-zero number, the SUT returns a 0x03 “Incorrect data” [16] response and the TCP connection is not terminated. The specification provides conflicting guidance on the expected behavior SUT behavior. Per the CIP Networks Library: Volume 2 EtherNet/IP Adaptation of CIP specification, “the receiver shall discard packets with a non-zero option field” [16]. The specification also says that the receiver shall not reject UnRegisterSession commands due to “unexpected values in the encapuslation header,” including non-zero Options and that the TCP connection shall be terminated [16].

In order to confirm the TCP session is not closed by UnRegisterSession request with an invalid Options field, an additional test is conducted. Following an UnRegisterSession command with a fuzzed Options field, a CIP Forward Open command is sent to the PLC. The SUT responds to the request with “Success” packet, confirming that the session remains open.

e. T13 Test Case

The MicroLogix complies with the ENIP requirement that an UnRegisterSession command sent over UDP shall be rejected with an 0x01 “Invalid or Unsupported” [16] error code.

3. ENIP SendRRData Results

There are multiple fields where the AB/RA MicroLogix’s implementation of the ENIP protocol deviates from the expected responses derived from the CIP Networks Library: Volume 2 EtherNet/IP Adaptation of CIP [16] specification. The expected and observed behaviors of each fuzzed field are discussed below.

a. T14 Test Case

The Session Handle is returned by the target in the ENIP Register Session reply packet, and is to be used in subsequent encapsulation commands within the ENIP session. When tested with session handles other than the valid handle of the ENIP session, the reply is not a 0x03 “Incorrect data” [16] error code as expected, but a successful service response. The CIP data in the response is identical to a message with a valid session handle (see Appendix A). However, the Wireshark protocol analyzer does not properly format the CIP Connection Manager data in replies to the invalid session handles. This is hypothesized to be a result of Wireshark attempting to match request/reply packet pairs with valid session handles.

b. T15 Test Case

The Status field indicates whether a receiver successfully executes a command. A zero response indicates success. Any other responses correlate to general error codes. According to the ENIP specification, the receiver must ignore all ENIP requests with a non-zero Status field, i.e., does not return a reply [16]. When testing the Status field of the SendRRData command, requests with Status fields between 0x00000000 and 0x0000FFFF are accepted and the SUT provides a successful ENIP-encapsulated CIP response with Status code 0x00000000. This deviates from the specification requirements [16]. When requests are sent with Status codes between 0x00010000 and 0xFFFFFFFF, the SUT performs as expected and provides no ENIP or ENIP-encapsulated CIP response. Only a TCP ACK packet is sent from the SUT to the fuzzer.

c. T16 Test Case

The Sender Context field allows a sender to place any data in the field. The receiver returns the same data in its response, which can be used by the sender to match requests with their replies [16]. For all tests, the returned values of this field match the expected values.

d. T17 Test Case

The Options field allows a sender to provide additional information about the command [16]. For the SendRRData Request, the specification dictates that the Options field be set to zero, and that the "receiver shall discard any packets with a non-zero option field" [16]. When tested with different non-zero options, the SUT returns successful replies, i.e., the returned status is 0x00000000.

e. T18 Test Case

The Interface Handle field identifies the intended communications interface of the command and must be set to zero for the SendRRData request [16]. When this field is set to a non-zero value, the SUT returns an ENIP response with the error code 0x03, as expected.

f. T19 Test Case

The Timeout field indicates the number of seconds the requested operation shall persist until it expires. When the field is set to zero, the timeout of the ENIP protocol assumes the timeout of the encapsulated protocol (CIP). When encapsulating CIP, the sender must set the Timeout field to zero and the receiver is to ignore the field [16]. The expected result for testing non-zero inputs in the Timeout field with CIP encapsulation is an ENIP response with the 0x03 "Incorrect data" [16] error code. However, the SUT returns successful ENIP-encapsulated CIP responses with the Timeout field set to 1024.

4. ENIP SendUnitData Results

Similar deviations from specification observed with SendRRData testing are also present in the SendUnitData testing.

a. T20 to T23 Test Cases

The fuzzing of Session Handle, Status, and Options fields demonstrate the same unexpected behavior observed in the SendRRData responses described above.

b. T24 Test Case

When the Interface Handle field is set to a non-zero value, the SUT returns an ENIP response with the 0x03 “Incorrect data” [16] error code, as expected.

c. T25 Test Case

The testing of the Timeout field shows unexpected behavior. The expected result for testing non-zero inputs in the Timeout field with CIP encapsulation is an ENIP response with a 0x03 “Incorrect data” [16] error code. However, the SUT returns successful ENIP-encapsulated CIP responses with the Timeout field set to zero. The unexpected SendUnitData responses are different than the unexpected SendRRData responses fuzzed under the same conditions. SendUnitData returns successful responses with the Timeout field set to zero, whereas SendRRData returns successful responses with the Timeout field set to 1024.

5. ENIP Reserved for Legacy Use Results

a. T26 and T27 Test Cases

The expected responses for the Legacy Use commands over TCP and UDP are a successful response, a 0x03 “Incorrect data” [16] response, or a 0x01 “Invalid or Unsupported” [16] response. Without knowledge of the packet structure for the Legacy Use commands, testing is limited to test packets that only include the individual Legacy Use command with no additional data attached. All commands sent over TCP return ENIP responses with the error code 0x03, except for the command code 0x01, which returns a successful ENIP response with the SUT’s IP address in the data field. The

Wireshark dissector recognized the commands 0x72 and 0x73 as Indicate Status and Cancel, respectively [16]. UDP-sent commands behave similarly to TCP-sent commands with regards to returning a successful response to command code 0x01. For all other command codes, UDP-sent commands returned the error code 0x01.

6. ENIP Reserved for Future Use Results

a. T28 and T29 Test Cases

Responses to the Future Use commands are expected to be an error code, either 0x03 or 0x01. The SUT returns the error code 0x03 status codes for TCP test cases and the error code 0x01 for UDP test cases.

B. CIP TEST RESULTS

The CIP Fuzzing tests do not cause any faults or disruption of service to the MicroLogix SUT. The test results indicate that MicroLogix does not support several of the tested commands. Table 10 summarizes both the expected and observed responses to the test cases.

Table 10. CIP Fuzz Testing Results

Test Number	CIP Command	Fuzzed Field	Expected Fuzzed Response	Actual Fuzzed Response
T30	Get_Attributes_All	Class	Class specific	Class specific See results below
T31	Get_Attributes_All	Instance	Attribute or Path destination unknown responses	Attribute or Path destination unknown responses
T32	Get_Attribute_List	Class	Attribute, Service not supported, or Path destination unknown responses	Service not supported or Path destination unknown responses
T33	Get_Attribute_List	Attribute_list	Attribute, Service not supported, or Path destination unknown responses	Service not supported or Path destination unknown responses

Test Number	CIP Command	Fuzzed Field	Expected Fuzzed Response	Actual Fuzzed Response
T34	Get_Attribute_List	Instance	Attribute, Service not supported, or Path destination unknown responses	Service not supported or Path destination unknown responses
T35	Get_Attribute_List	Attribute_count	Error status or no response for Attribute_count fields exceeding maximum allowable	TCP ACK for values greater than 223 Attributes in Attribute_count
T36	Get_Attribute_Single	Class	Class specific	Class specific See results below
T37	Get_Attribute_Single	Instance	Attribute not supported or Service not supported	Service not supported, Attribute not supported, or Path destination unknown
T38	Get_Attribute_Single	Attribute	Service not supported	Service not supported or Attribute not supported
T39	Find_Next_Object_Instance	Class	Service not supported or Path destination unknown	Service not supported or Path destination unknown
T40	Find_Next_Object_Instance	Instance	Service not supported or Path destination unknown	Service not supported or Path destination unknown
T41	Find_Next_Object_Instance	Maximum Returned Values	Service not supported	Service not supported

1. CIP Get_Attributes_All Results

To determine a baseline MicroLogix response for the Get_Attributes_All command, a packet with Class 0x01 (Identity) and Instance 0x01 is sent to the SUT. All CIP devices are required to support Instance 0x01 of the Identity Object [15]. The MicroLogix returns a successful CIP response with the seven required attributes: Vendor ID, Device Type, Product Code, Major and Minor Revisions, Status, Serial Number, and

Product Name (see Table 11). The thirteen optional or conditional attributes defined in the specification are not observed in the MicroLogix responses [15].

Table 11. Identity Object Instance Attributes. Adapted from [15].

Attr ID	Need in Implem	Access Rule	NV	Name	Data Type	Description of Attribute
1	Required	Get	NV	Vendor ID	UINT	Identification of each vendor by number
2	Required	Get	NV	Device Type	UINT	Indication of general type of product
3	Required	Get	NV	Product Code	UINT	Identification of a particular product of an individual vendor
4	Required	Get	NV	Revision	STRUCT of:	Revision of the item the Identity Object represents
				Major Revision	USINT	
				Minor Revision	USINT	
5	Required	Get	V	Status	WORD	Summary status of device
6	Required	Get	NV	Serial Number	UDINT	Serial number of device
7	Required	Get	NV	Product Name	SHORT_ STRING	Human readable identification

a. *T30 Test Case*

When the Class field is fuzzed with values from 0x0 to 0xFF, with Instance 0x01, the following behavior was observed:

- Three different Class field inputs between 0x00 and 0xFF return successful CIP packets with attribute information: 0x01 (Identity), 0xF5 (TCP/IP Interface), and 0xF6 (Ethernet Link).
- Three Class field inputs return General Status 0x08 “Service not supported” [15] responses: 0x02 (Message Router), 0x06 (Connection Manager), and 0x67 (PCCC Object).
- The remaining Class inputs return CIP responses with General Status 0x05 “Path destination unknown” [15]. This code is used when the target device does not recognize a class, instance or structure element in the object’s request [15].

b. T31 Test Case

When fuzzing the Instance field with values from 0x00 to 0xFF with the Class field set to 0x01, the SUT responds successfully to two Instance inputs: 0x00 and 0x01. The responses for all other Instance inputs indicate a General Status 0x05 “Path destination unknown” [15]. This is an expected response.

The Instance 0x00 is handled as a special case because it references the Class instead of a particular Instance within the class [15]. Therefore, the response of the Instance 0x00 is at the Class level as shown in Table 12.

Table 12. Identity Object Get_Attributes_All Response for Instance 0x00. Source: [15].

Attribute ID	Data Type	Attribute Name	Default Value (if not implemented)
1	UINT	Revision	1
2	UINT	Max Instance	1
6	UINT	Max ID Number Class Attributes	0
7	UINT	Max ID Number Instance Attributes	0

The other successful response, Instance 0x01, is used as a baseline command and is previously explained.

2. CIP Get_Attribute_List Results

To determine baseline functionality, a request is sent to the SUT with the following parameters: Class 0x01, Instance 0x01, and Attribute 0x01. There are two possible expected SUT responses. If the SUT supports the command, it is to respond with the requested Attribute (Vendor ID) information. However, since Get_Attribute_List is an optionally supported command at the Class and Instance level [15], the SUT may not provide the requested Attribute response. When tested, the MicroLogix responds with a General Status 0x08 “Service not supported” [15] packet.

a. *T32 Test Case*

To test the Class field with values 0x00 to 0xFF, the Instance and Attribute fields are set to 0x01 while Class is fuzzed. The SUT returns a General Status 0x08 “Service not supported” [15] CIP response for six of the Class field inputs: 0x01 (Identity), 0x02 (Message Router), 0x06 (Connection Manager), 0x67 (PCCC Object), 0xF5 (TCP/IP Interface), and 0xF6 (Ethernet Link). All other responses have a General Status of 0x05 “Path destination unknown” [15].

b. *T33 Test Case*

While testing the Instance field with values 0x00 to 0xFF, the Class and Attribute fields are set to 0x01. Only Instances 0x00 and 0x01 return General Status 0x08 “Service not supported” [15] responses. All other tested Instances return General Status 0x05 “Path destination unknown” [15] responses.

c. *T34 Test Case*

When fuzzing the Attribute field with the Class and Instance fields set 0x01, the SUT returns General Status 0x08 “Service not supported” [15] responses for each Attribute tested. The tested Attribute values are 0x00 to 0xFF.

d. *T35 Test Case*

To determine the effects of exceeding the maximum number of attributes that can be requested, packets with increasing Attribute_Count are sent to the SUT. Attribute IDs 1 through 7 are utilized and repeated due to their observed presence from the Get_Attributes_All response previously conducted. The SUT returns CIP responses with Status 0x08 “Service not supported” [15] for Get_Attribute_List requests with Attribute_Counts from 0 to 223. When the SUT receives a Get_Attribute_List request with an Attribute_Count of 224 or greater, it does not send a CIP response. The SUT sends only a TCP ACK in response.

3. CIP Get_Attribute_Single Results

The Get_Attribute_Single request is an optional command and thus it is hypothesized that the response would be a 'Service not supported" [15] message. Per the specification [15], the Identity Object only supports this command if Class Attributes are implemented. The observed response from the Get_Attributes_All tests for the Identity Object with Instance 0x00 and Attribute 0x01 return default values, indicating no Class Attributes are set for the Identity Object.

a. T36 Test Case

To test the Class field, the Instance field is set to 0x00 and the Attribute field is set to 0x01. The SUT returns a "Service not supported" CIP response for six of the Class field inputs: 0x01 (Identity), 0x02 (Message Router), 0x06 (Connection Manager), 0x67 (PCCC Object), 0xF5 (TCP/IP Interface), and 0xF6 (Ethernet Link). All other responses have a General Status 0x05 "Path destination unknown" [15].

b. T37 Test Case

When testing the Instance field, the Class and Attribute fields are set to 0x01. The SUT returns "Service not supported" messages when it receives request packets with the Instance field set to 0x00. When it receives requests with Instance 0x01, the SUT returns the General Status 0x14 "Attribute not supported" [15] messages. All other Instances returned General Status 0x05 "Path destination unknown" [15] responses.

c. T38 Test Case

To test the Attribute field, packets are sent with the Class and Instance fields set to 0x01. The SUT responds to all fuzzed Attribute inputs with the General Status 0x14 "Attribute not supported" [15] messages.

4. CIP Find_Next_Object_Instance Results

In order to establish baseline behavior for the Find_Next_Object_Instance request, test packets with Class 0x01 (Identity) and Instance 0x00 fields are sent to the SUT. The Identity Object conditionally supports the command if non-consecutive

Instances exit [15]. From the Get_Attributes_All test using the Identity Object, no non-consecutive Instances are observed. Therefore, our expected and observed behavior of the SUT is to return a General Status 0x08 “Service not supported” [15] message.

a. T39 Test Case

To test the Class field, the Instance field is set to 0x00 while Class is fuzzed. The SUT returns a General Status 0x08 “Service not supported” [15] CIP response for six of the Class field inputs: 0x01 (Identity), 0x02 (Message Router), 0x06 (Connection Manager), 0x67 (PCCC Object), 0xF5 (TCP/IP Interface), and 0xF6 (Ethernet Link). All other responses have a General Status 0x05 “Path destination unknown” [15].

b. T40 Test Case

When testing the Instance field, Class is set to 0x01. Requests with Instance 0x00 and 0x01 return General Status 0x08 “Service not supported” [15] responses. All other fuzzed Instance inputs return General Status 0x05 “Path destination unknown” [15] messages.

c. T41 Test Case

To test the Maximum Returned Values field, Class is set to 0x01 and Instance is set to 0x00. The Maximum Returned Values field is tested with inputs between 0x00 and 0xFF. All requests return General Status 0x08 “Service not supported” [15] responses.

C. PCCC TEST RESULTS

The PCCC tests do not cause any faults or disruption of service to the MicroLogix 1100 (T42-T72) or ControlLogix 1756-L71 (T73) SUTs. Table 13 summarizes both expected and observed responses to the test cases. The N/A indicator in the Fuzzed Field column indicates the command is functionally tested only.

Table 13. PCCC Fuzz Testing Results

Test Number	PCCC Command	Fuzzed Field	Expected Fuzzed Response	Actual Fuzzed Response
MicroLogix Tests				
T42	Echo	Data: 0 bytes	Response with 0 bytes attached	Response with 0 bytes attached
T43	Echo	Data: Max Length	243-byte maximum	247-byte maximum
T44	Echo	Data: 8 bytes	Response with 8 bytes attached	Response with 8 bytes attached
T45	Echo	Data: 9 bytes	Response with 9 bytes attached	Response with 9 bytes attached
T46	Echo	Data: 10 bytes	Response with 10 bytes attached	Response with 10 bytes attached
T47	Echo	Data: 40 bytes	Response with 40 bytes attached	Response with 40 bytes attached
T48	Echo	Data: 243 bytes	Response with 243 bytes attached	Response with 243 bytes attached
T49	Echo	Data: Maximum bytes returned by module with no errors	Response with same number of bytes attached as request	Response with 247 bytes attached
T50	Echo	Data: 248 bytes	Response with error message	“Routing failure, request packet too large” [17] response
T51	Echo	Data: 256 bytes	Response with error message	“Routing failure, request packet too large” [17] response
T52	Protected Typed File Read	Size	Response with requested data or error message	“illegal command or format” [17] response
T53	Protected Typed File Read	Tag	Response with requested data or error message	“illegal command or format” [17] response
T54	Protected Typed File Read	Offset	Response with requested data or error message	“illegal command or format” [17] response

Test Number	PCCC Command	Fuzzed Field	Expected Fuzzed Response	Actual Fuzzed Response
T55	Protected Typed File Read	File Type	Response with requested data or error message	“illegal command or format” [17] response
T56	Protected Typed File Write	Size	Response with no errors or error message	“illegal command or format” [17] response
T57	Protected Typed File Write	Tag	Response with no errors or error message	“illegal command or format” [17] response
T58	Protected Typed File Write	Offset	Response with no errors or error message	“illegal command or format” [17] response
T59	Protected Typed File Write	File Type	Response with no errors or error message	“illegal command or format” [17] response
T60	Protected Typed File Write	Data	Response with no errors or error message	“illegal command or format” [17] response
T61	Protected Typed Logical Write with Three Address Fields	Size	Response with no errors or error message	“illegal command or format” [17] or “access denied, improper privilege” [17] responses
T62	Protected Typed Logical Write with Three Address Fields	File No.	Response with no errors or error message	“illegal command or format” [17] response
T63	Protected Typed Logical Write with Three Address Fields	File Type	Response with no errors or error message	“illegal command or format” [17] response
T64	Protected Typed Logical Write with Three Address Fields	Element No.	Response with no errors or error message	“illegal command or format” [17] response
T65	Protected Typed Logical Write with Three Address Fields	Sub-Element No.	Response with no errors or error message	“illegal command or format [17] response
T66	Unprotected Read	Address	Response with requested data or error message	“illegal command or format” [17] response

Test Number	PCCC Command	Fuzzed Field	Expected Fuzzed Response	Actual Fuzzed Response
T67	Unprotected Read	Size	Response with requested data or error message	“illegal command or format” [17] response
T68	Diagnostic Status-Functionality Test	N/A	Diagnostic Status information response	Diagnostic Status information response
T69	Read Diagnostic Counters	Address	Response with requested data or error message	“illegal command or format” [17] response
T70	Read Diagnostic Counters	Size	Response with requested data or error message	“illegal command or format” [17] response
T71	Restart-Functionality Test	N/A	Response with error message	“illegal command or format” [17] response
T72	Download Completed-Functionality Test	N/A	Response with error message	“access denied, improper privilege” [17] response
ControlLogix Tests				
T73	Protected Typed Logical Read with Three Address Fields	File No., File Type	SUT Fault	No fault. EXT STS “Address doesn’t point to something usable” [17] response

1. PCCC Echo Results

a. T42 to T51 Test Cases

The SUT returns successful responses to properly formatted PCCC Echo requests. The data specified in Echo requests, up to 247 bytes, are successfully transmitted back to the fuzzer in a CIP-encapsulated response packet. The observed 247-byte limit exceeds the maximum of 243 data bytes indicated in the specification [17]. When Echo commands are transmitted with greater than 247 data bytes attached, the SUT returns a CIP-encapsulated response indicating “Routing failure, request packet too large.”

2. PCCC Protected Typed File Read Results

a. T52-T55 Test Cases

The SUT responds uniformly to all fuzzed Size, Tag, Offset, and File Type field inputs by returning a STS 0x10 “illegal command or format” [17] code.

3. PCCC Protected Typed File Write Results

a. T56-T60 Test Cases

The SUT responds to all fuzzed Size, Tag, Offset, File Type, and Data field inputs by returning a STS 0x10 “illegal command or format” [17] code.

4. PCCC Protected Logical Write with Three Address Fields Results

a. T61 Test Case

When fuzzing the Byte Size field of the command, all inputs except 0x00 return successful CIP-encapsulated PCCC packets with a STS 0x10 “illegal command or format” [17] code. When the Byte Size field is set to 0x00, the STS field returns 0xF0, indicating an EXT STS is appended. The returned EXT STS byte is 0x0B, indicating “access denied, improper privilege” [17].

b. T62-T65 Test Cases

The SUT responses uniformly to all fuzzed File Number, File Type, Element Number, and Sub-Element Number field inputs by returning a STS 0x10 “illegal command or format” [17] code.

5. PCCC Unprotected Read Results

a. T66-T67 Test Cases

The SUT responses uniformly to all fuzzed Address and Size field inputs by returning a STS 0x10 “illegal command or format” [17] code.

6. PCCC Diagnostic Status Results

a. T68 Test Case

The Diagnostic Status command returns a successful CIP-encapsulated PCCC response. The specification [17] states that the MicroLogix 1000's response is 24 bytes [17]. The MicroLogix 1100 returns 25 bytes of data. Due to this difference, it is not possible to determine the exact meaning of the returned byte values. It appears that the returned data provides information on the SUT's system status as well as an ASCII representation that displays the SUT's model information: 1763-LEC.

7. PCCC Read Diagnostic Counters Results

a. T69 Test Case

When fuzzing the Address field of the Read Diagnostic Counters command, the SUT returns a CIP-encapsulated PCCC response with a STS 0x10 "illegal command or format" [17] code, for all cases except when the Address field is set to 0x0000. During testing, the Size field is constant at 0x01.

b. T70 Test Case

When fuzzing the Size field, the SUT responds with the requested number of bytes when the Size inputs are below 0x6D. These responses contain bytes with zero and non-zero values. The SUT responds to any input of 0x6D or greater with a packet containing no returned data and a STS 0x10 "illegal command or format" [17] code.

8. PCCC Restart Results

a. T71 Test Case

Responses to the Restart command functionality test have STS 0x10 "illegal command or format" [17] codes.

9. PCCC Download Completed Results

a. T72 Test Case

The SUT responds to the Download Completed command with an EXT STS 0x0B “access denied, privilege violation” [17] code.

10. PCCC Protected Logical Read with Three Address Fields on ControlLogix Results

a. T73 Test Case

We speculate that the MicroLogix vulnerability related to this command [9] would be present in the more advanced ControlLogix PLC due to the common practice of reusing legacy code without proper testing in different products from the same manufacturer. Fuzzing the File No. and File Type fields of the Protected Logical Read with Three Address Fields does not produce a fault in the ControlLogix, as observed in the MicroLogix. This proves our hypothesis false.

There is an observable difference between the MicroLogix and ControlLogix responses to the command when File No. and File Type are fuzzed. From previous testing [1], we observe that MicroLogix responds in one of five ways: 1) responds with an STS 0x10 “illegal command or format” code, 2) responds with an EXT STS 0x0B “Access denied, improper privilege” [17] code, 3) responds with an EXT STS 0x0C “condition cannot be generated, resource is not available” [17] code, 4) responds with data, or 5) responds by entering a fault condition [17]. Table 14 illustrates sample request packet field contents and the range of SUT responses.

Table 14. Example MicroLogix 1100 Responses to PCCC Protected Logical Read with Three Address Fields Command

Byte Size	File Type	File No.	Element No.	Sub-element No.	SUT Response
0x01	0x10	0xD0	0x84	0x00	STS 0x10
0x57	0x75	0x65	0x10	0x00	EXT STS 0x0B
0x56	0xBD	0x4C	0x59	0x00	EXT STS 0x0C
0x1C	0x2A	0x62	0x01	0x00	Data response
0xC8	0x03	0x47	0xBC	0x00	Fault response

In all tests, the ControlLogix SUT returns a STS 0xF0 “Error code in the EXT STS byte” code and an EXT STS byte of 0x06 “Address doesn’t point to something usable” [17]. This difference in SUT responses may be a useful tool in fingerprinting the manufacturer and model of a target PLC.

D. DISCUSSION

Our fuzz testing does not uncover any MicroLogix 1100 vulnerabilities. However, we observe some deviations from the expected responses in the MicroLogix implementation of ENIP and PCCC protocols. No CIP deviations are observed. Multiple optional tested CIP commands are not supported by MicroLogix 1100 PLCs. Table 15 provides a summary of the discovered MicroLogix unexpected responses.

Table 15. Summary of MicroLogix 1100Unexpected Responses

Test Number	Command	Fuzzed Field	Protocol	Expected Fuzzed Response	Deviation Response
ENIP Tests					
T7	List Services/Identity/Interfaces	Options	TCP	Packet discarded	Error 0x03 response
T8	List Services/Identity/Interfaces	Options	UDP	Packet discarded	Error 0x03 response
T9	UnRegisterSession	Session Handle	TCP	Error 0x03 TCP close	Error 0x03 response No TCP close
T10	UnRegisterSession	Status	TCP	Error 0x03 TCP close	Error 0x03 response No TCP close
T12	UnRegisterSession	Options	TCP	Error 0x03 TCP close	Error 0x03 response no TCP close
T14	SendRRData	Session Handle	TCP	Error 0x03	No error, Successful response
T15	SendRRData	Status	TCP	TCP ACK	Successful responses for Status fields between 0x00000000 and 0x0000FFFF

Test Number	Command	Fuzzed Field	Protocol	Expected Fuzzed Response	Deviation Response
T17	SendRRData	Options	TCP	TCP ACK	Successful response
T19	SendRRData	Timeout	TCP	Error 0x03	Successful response, Timeout field 1024
T20	SendUnitData	Session Handle	TCP	Error 0x03	No error, Successful response
T21	SendUnitData	Status	TCP	TCP ACK	Successful responses for Status fields between 0x00000000 and 0x0000FFFF
T23	SendUnitData	Options	TCP	TCP ACK	Successful response
CIP Tests					
No observed deviations from specification: Tested optional commands not implemented by MicroLogix 1100					
PCCC Tests					
T43	PCCC Echo	Data: Max Length	TCP	243-byte maximum	247-byte maximum
T52-T55	Protected Typed File Read	Size, Tag, Offset, File Type	TCP	Data response	“illegal command or format” [17] response
T56-T60	Protected Typed File Write	Size, Tag, Offset, File Type, Data	TCP	Data response	“illegal command or format” [17] response
T61-T65	Protected Typed Logical Write with Three Address Fields	Size, File No., File Type, Element No. Sub-Element No.	TCP	Response with no errors or error message	“illegal command or format” [17] response
T66-T67	Unprotected Read	Address, Size	TCP	Data response	“illegal command or format” [17] response
T68	Diagnostic Status	N/A, Functionality Test	TCP	24-byte Diagnostic Status information response	25-byte Diagnostic Status information response

The deviations in the ENIP implementation may be the result of manufacturer implementation decisions. A potential explanation for the PCCC deviations is that the reference specification [9] applies to the MicroLogix 1000 model. While we expect the implementation to be similar between the 1000 and 1100 models, there are differences in processing capability, memory allocations, and functionality between the PLCs, which may account for the deviations.

Our ControlLogix testing disproves the hypothesis that the PCCC Protected Typed Logical Read with Three Address Fields vulnerability in MicroLogix 1100 also affects the ControlLogix 1756-L71. In contrast to the fault condition observed on the MicroLogix 1100, the ControlLogix 1756-L71 returns an error message upon receiving a request with the File No. field ranges between 0x2 to 0x8 and the File Type is 0x47 or 0x48. Table 16 illustrates the ControlLogix response.

Table 16. Summary of ControlLogix 1756-L71 Response Deviations

Test Number	Command	Fuzzed Field	Protocol	Expected Fuzzed Response	Deviation Response
T73	Protected Typed Logical Read with Three Address Fields	File No. (0x02-0x08), File Type (0x47 or 0x48)	TCP	SUT Fault	No fault. EXT STS 0x06 “Address doesn’t point to something usable” [1] response

The deviations may provide useful information for application-layer fingerprinting of PLC devices. By cataloging the unique responses returned from the MicroLogix 1100 and ControlLogix 1756-L71, we can begin compiling a corpus of PLC response signatures. This can be used to classify PLC modules through traffic analysis.

VI. CONCLUSION AND FURTHER WORK

A. SUMMARY

Motivated by the increasing employment of industrial control systems on U.S. Navy vessels and the potential for vulnerabilities in the utilized communication protocols, we aim to test the implementation of industrial network protocols on a PLC. Two hypotheses drive our testing. The first hypothesis is that undiscovered software flaws existed in the implementation of ENIP, CIP, and PCCC protocols used by the MicroLogix PLCs. The second hypothesis is that network vulnerabilities known to exist in older PLCs help inform on the robustness of the ICS network stack in more modern PLCs.

To verify our hypotheses, we use a fuzz testing methodology to stress test selected fields in target commands and monitor the system responses. To accomplish this, we use the Scapy-based ENIP Fuzz program [9] and modify the code to expand the range of testable protocol commands. We test our first hypothesis on the MicroLogix 1100 PLC by selecting a range of commands from the ENIP, CIP, and PCCC protocols that were not previously tested and systematically fuzzed the modifiable fields. Candidate protocol commands are evaluated for fuzzing based on their likelihood of creating a fault condition while not permanently damaging the test PLC or corrupting the functionality of the MicroLogix system.

The results of our fuzz testing do not uncover any new vulnerabilities in the MicroLogix 1100 PLC. However, we observe several unexpected responses in four ENIP commands (List Services/Identity/Interfaces, UnRegisterSession, SendRRData, and SendUnitData), and six PCCC commands (Echo, Protected Typed File Read, Protected Typed File Write, Protected Logical Write with Three Address Fields, Unprotected Read, and Diagnostic Status).

Our second hypothesis is tested by sending to the ControlLogix 1756-L71 specially crafted PCCC Protected Logical Read with Three Address Fields packets that trigger a fault condition in the MicroLogix 1100 [9]. By replicating the fault-inducing packet configuration of the command and applying it to a more advanced PLC, we aim to test if cross-generational vulnerabilities existed in AB/RA PLCs.

Instead of entering a fault state, the ControlLogix 1756-L71 PLC returns an error message upon receiving the fault-inducing test packets. This behavior disproves our hypothesis that the same MicroLogix 1100 vulnerability would affect the ControlLogix 1756-L71 PLC.

B. FUTURE WORK

In addition to PLC fingerprinting, the unique SUT responses observed during our testing may also be used by an intrusion detection system to catch malicious probing activities. To provide a larger context and differentiation among various PLCs, we plan to perform additional EtherNet/IP fuzz testing on the ControlLogix 1756-L71 and other ControlLogix models. These tests will provide insights on whether the observed response to the PCCC Protected Logical Read with Three Address Field command is specific to that command or is common to all PCCC requests, and on whether the PCCC support is the same or different across ControlLogix models.

Another extension to this work is to test the MicroLogix 1000 PLC to determine if the deviations observed in the MicroLogix 1100 are unique to that model or if the MicroLogix family uses a different implementation than detailed in the specification [17].

The scope of this thesis focuses on two different generations of AB/RA PLCs and the EtherNet/IP protocol suite. The ENIP Fuzz program can be enhanced to support other industrial protocols such as PROFINET or DNP3. The enhancement will provide a flexible test platform, which can be used to perform penetration testing, intrusion detection, and fingerprinting reconnaissance on a wide range of industrial control systems.

APPENDIX A. ENIP COMMAND RESPONSES

The following Wireshark captures in Figures 56–93 illustrate test case responses for each command. For certain test cases, the corresponding request command sent to the SUT is also included to show how select fuzzed field inputs affect SUT responses. For descriptions of SUT responses, see Chapter V: Test Analysis.

A. ENIP LISTSERVICES TEST CASES

This section shows the results of the ENIP ListServices test cases.

(1) T1 Results

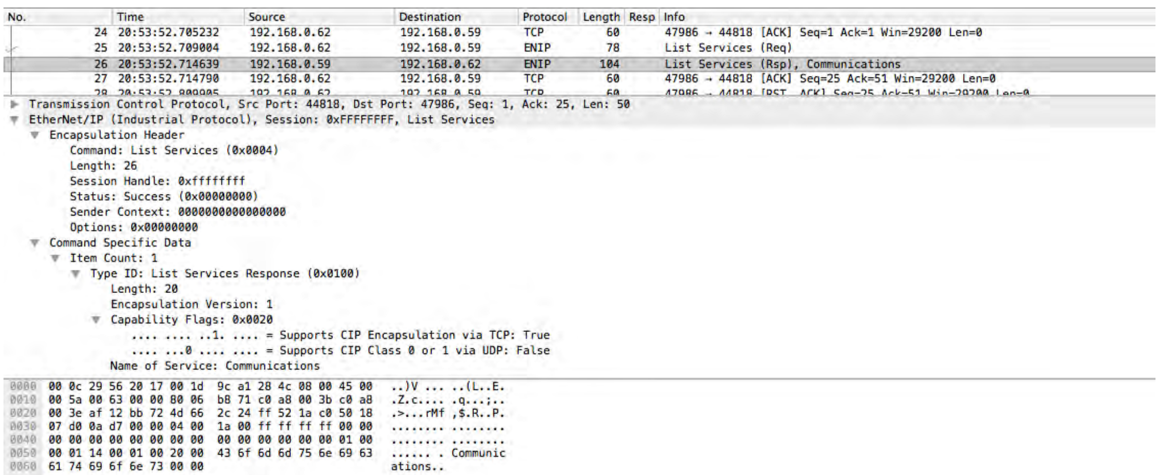


Figure 56. ListServices Response over TCP (Fuzzed Session Handle)

(2) T2 Results

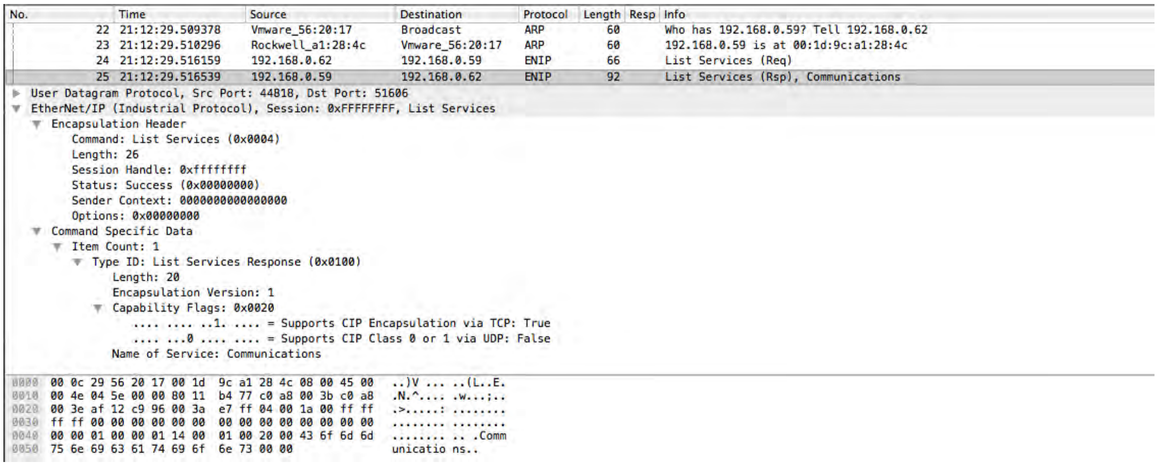


Figure 57. ListServices Response over UDP (Fuzzed Session Handle)

(3) T3 Results

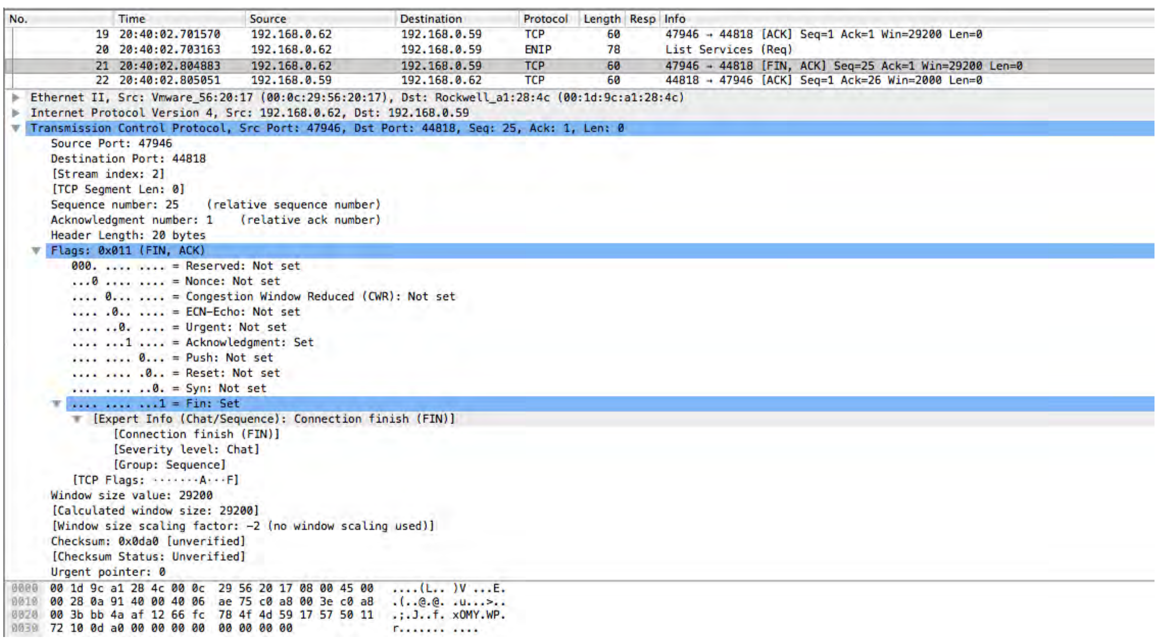


Figure 58. ListServices Response over TCP (Fuzzed Status)

(4) T4 Results

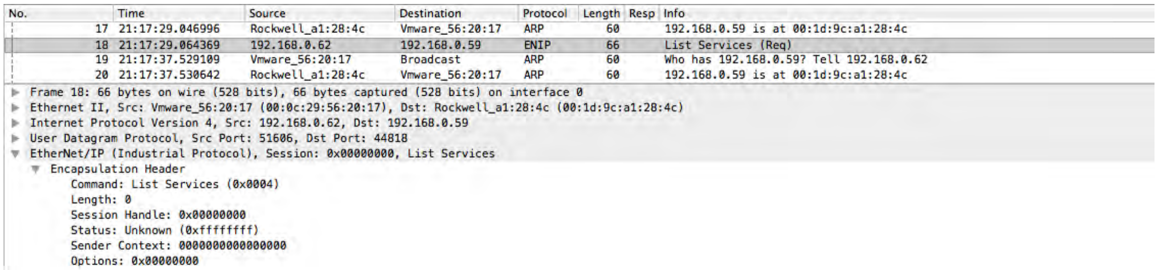


Figure 59. ListServices Response over UDP (Fuzzed Status)

(5) T5 Results

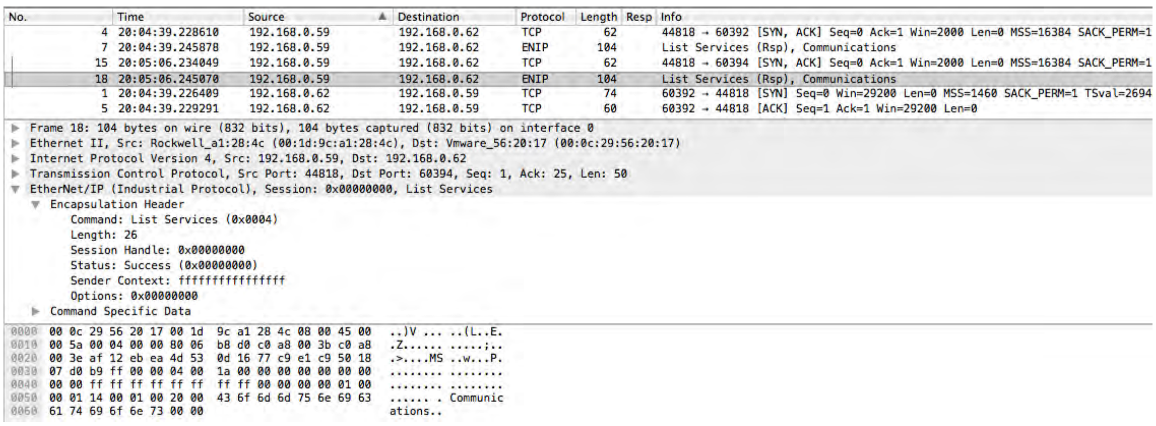


Figure 60. ListServices Response over TCP (Fuzzed Sender Context)

(6) T6 Results

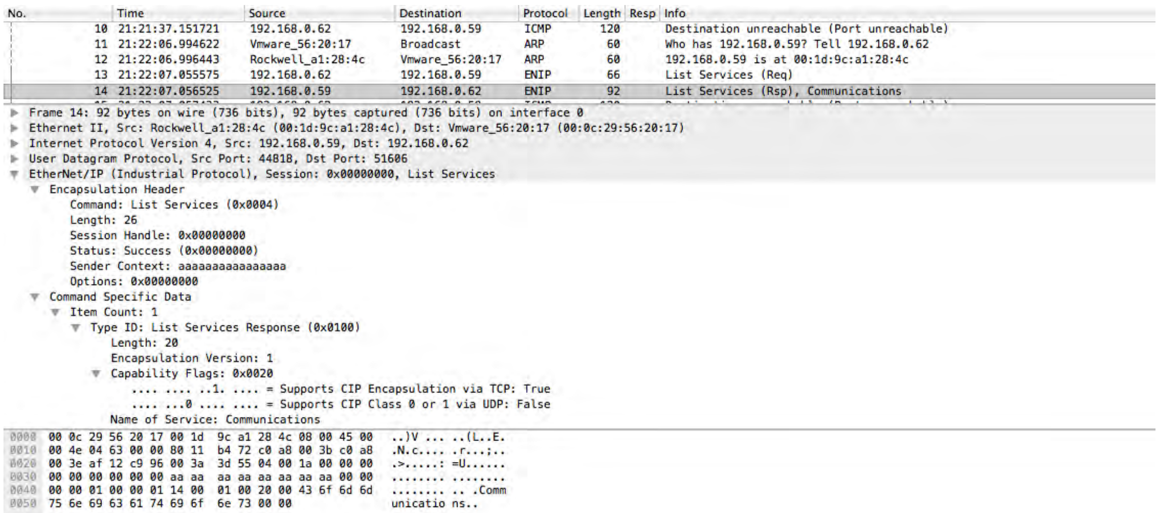


Figure 61. ListServices Response over UDP (Fuzzed Sender Context)

(7) T7 Results

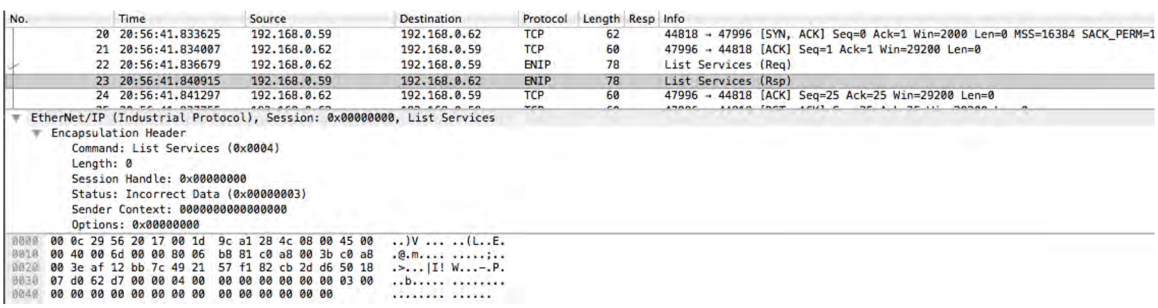


Figure 62. ListServices Response over TCP (Fuzzed Options)

(8) T8 Results

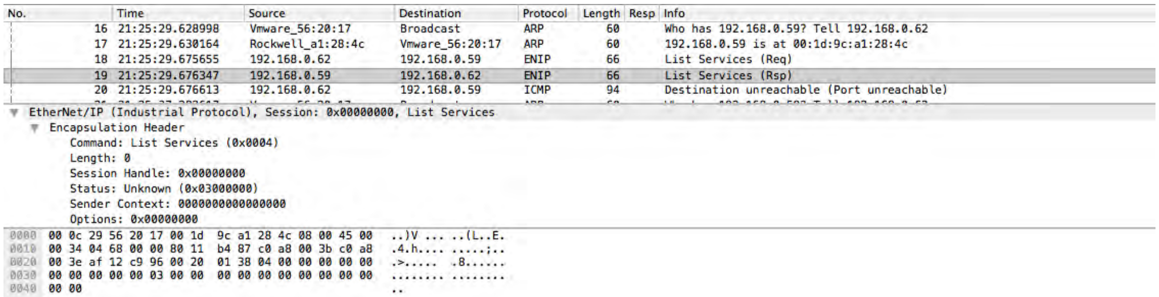


Figure 63. ListServices Response over UDP (Fuzzed Options)

B. ENIP UNREGISTERSESSION TEST CASES

This section shows the results of the ENIP UnRegisterSession test cases.

(1) T9 Results

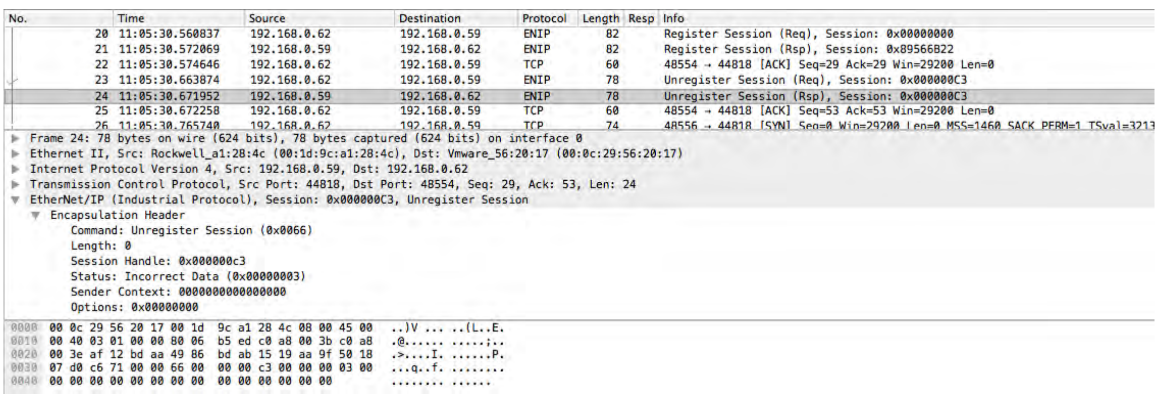


Figure 64. UnRegisterSession Response over TCP (Fuzzed Session Handle)

(2) T10 Results

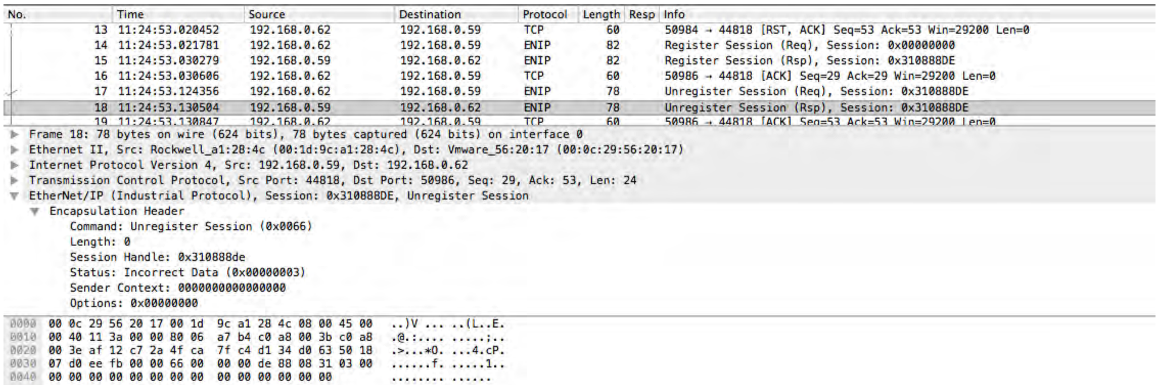


Figure 65. UnRegisterSession Response over TCP (Fuzzed Status)

(3) T11 Results

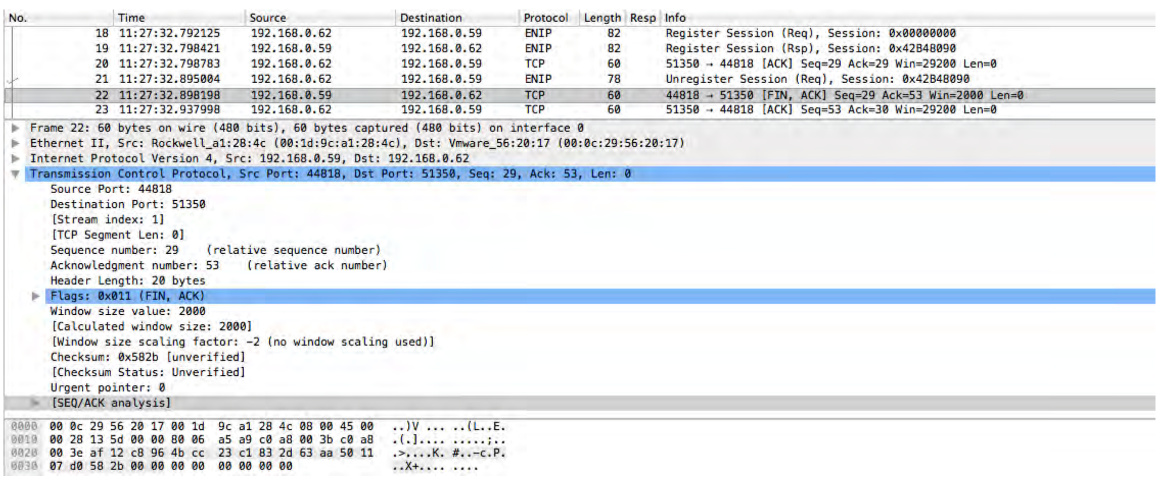


Figure 66. UnRegisterSession Response over TCP (Fuzzed Sender Context)

(4) T12 Results

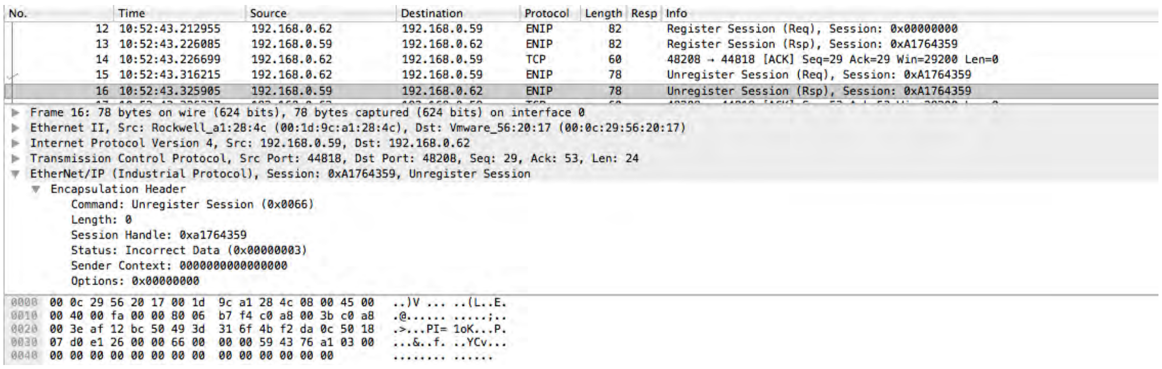


Figure 67. UnRegisterSession Response over TCP (Fuzzed Options)

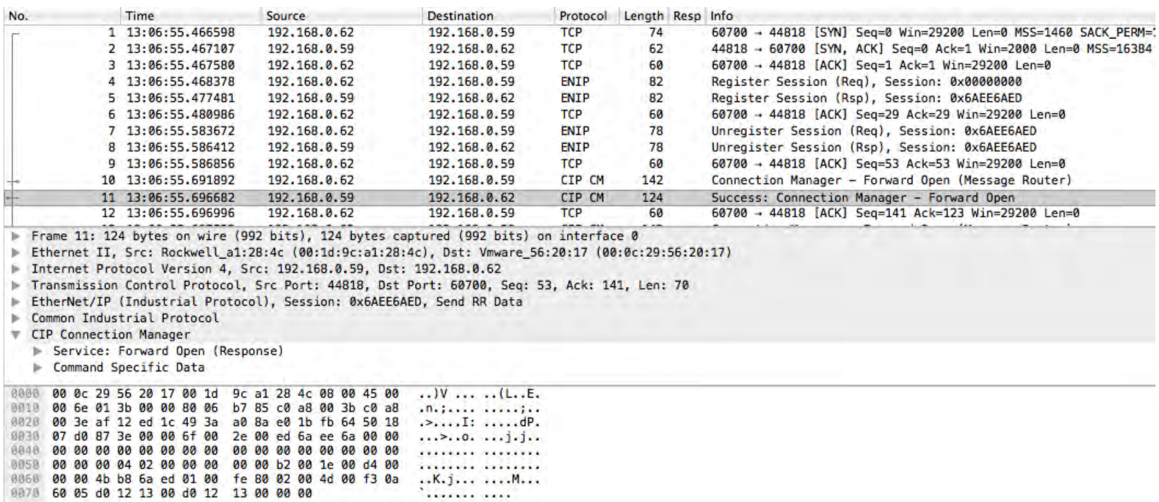


Figure 68. CIP Forward Open Response Following ENIP UnRegisterSession Request with Fuzzed Options Field

(5) T13 Results

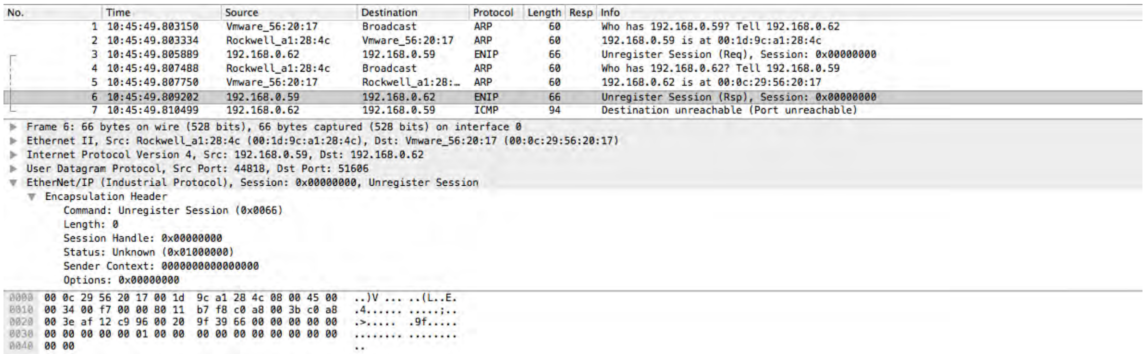


Figure 69. UnRegisterSession Response over UDP (Functionality Test)

C. ENIP SENDRRDATA TEST CASES

This section shows the results of the ENIP SendRRData test cases.

(1) T14 Results

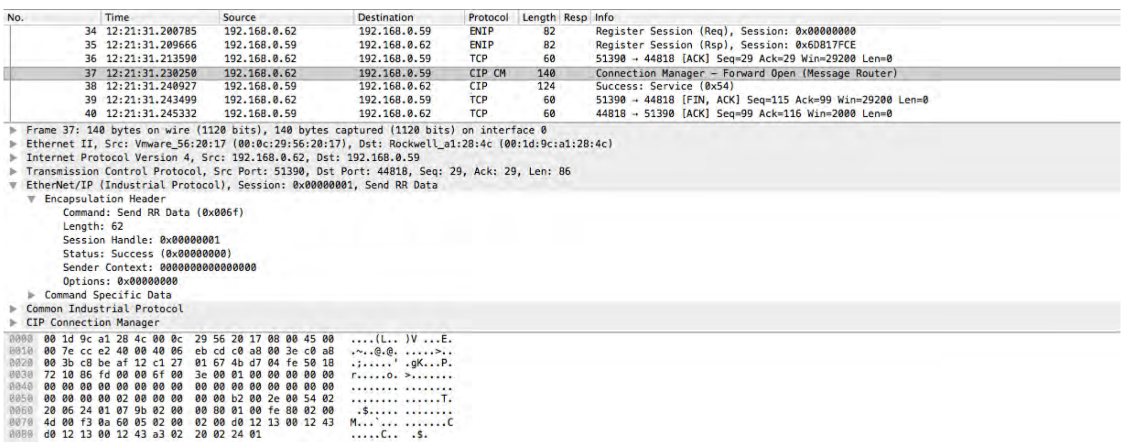


Figure 70. SendRRData Request over TCP (Fuzzed Session Handle)

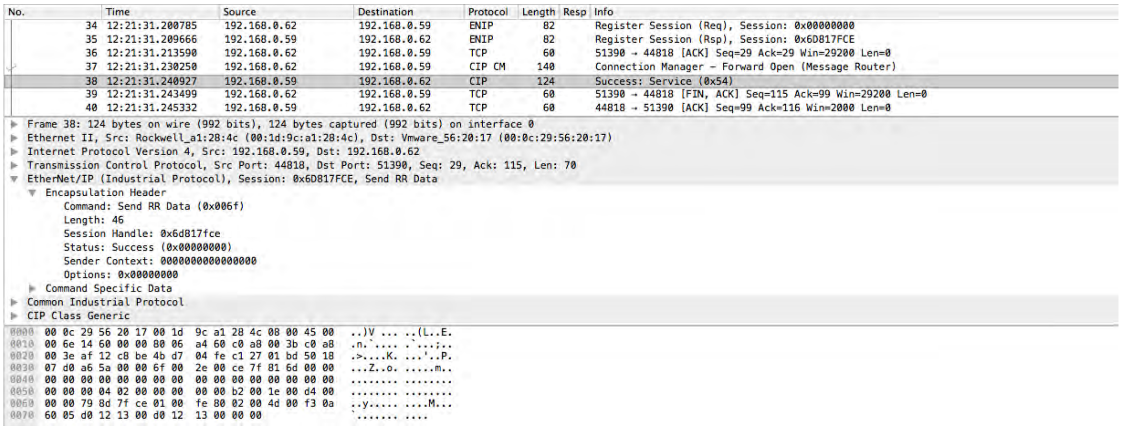


Figure 71. SendRRData Response over TCP (Fuzzed Session Handle)

(2) T15 Results

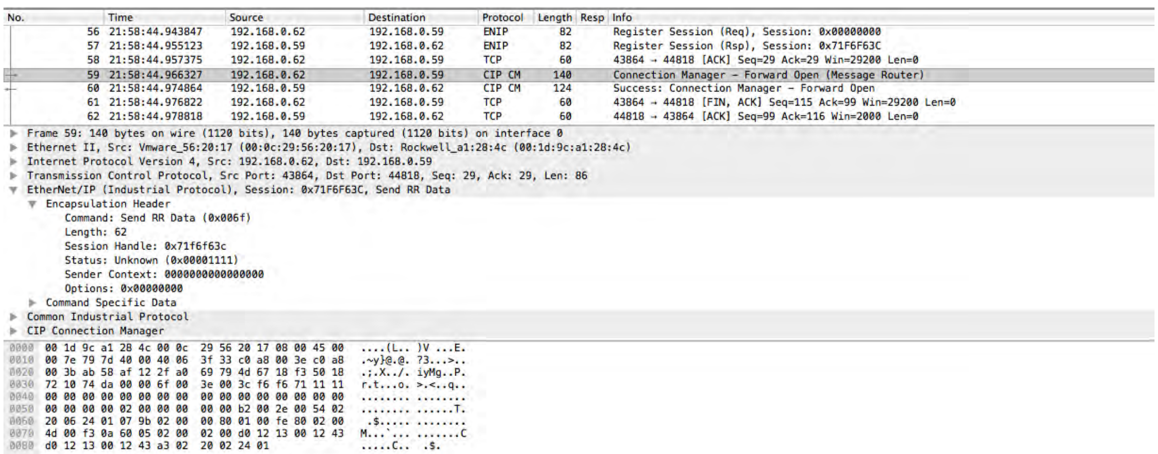


Figure 72. SendRRData Request over TCP (Fuzzed Status)

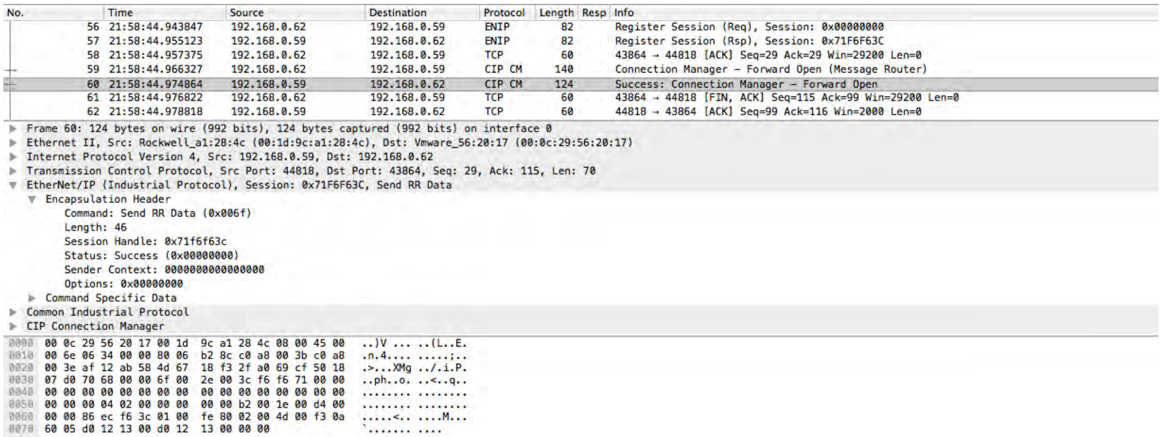


Figure 73. SendRRData Response over TCP (Fuzzed Status)

(3) T16 Results

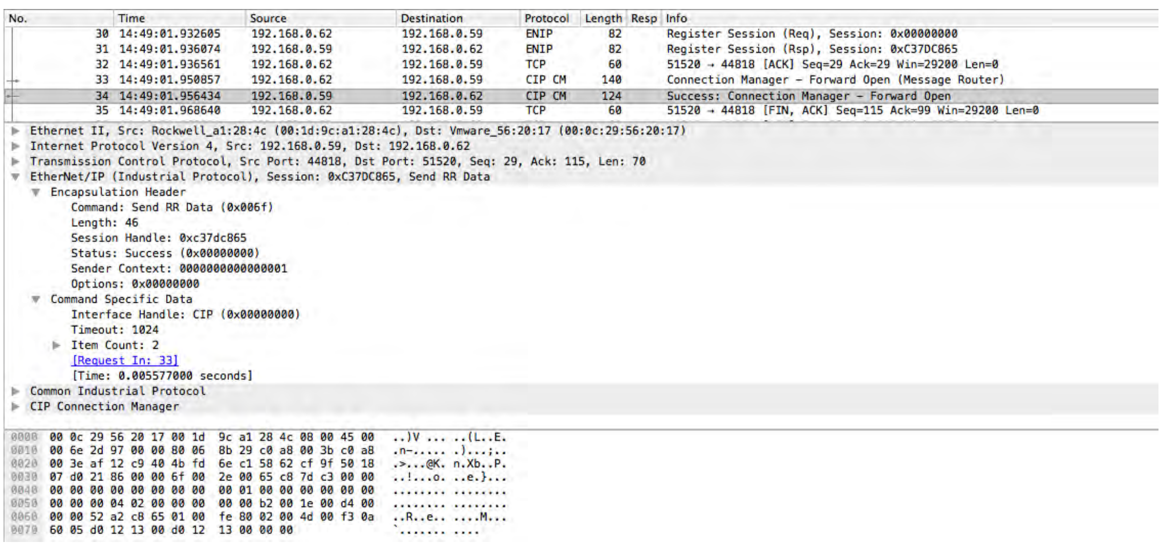


Figure 74. SendRRData Response over TCP (Fuzzed Sender Context)

(4) T17 Results

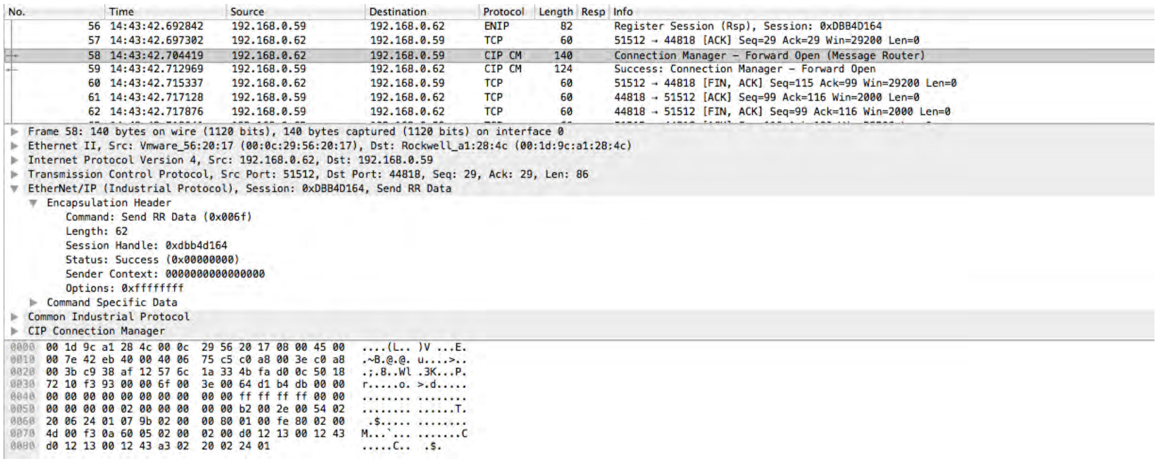


Figure 75. SendRRData Request over TCP (Fuzzed Options)

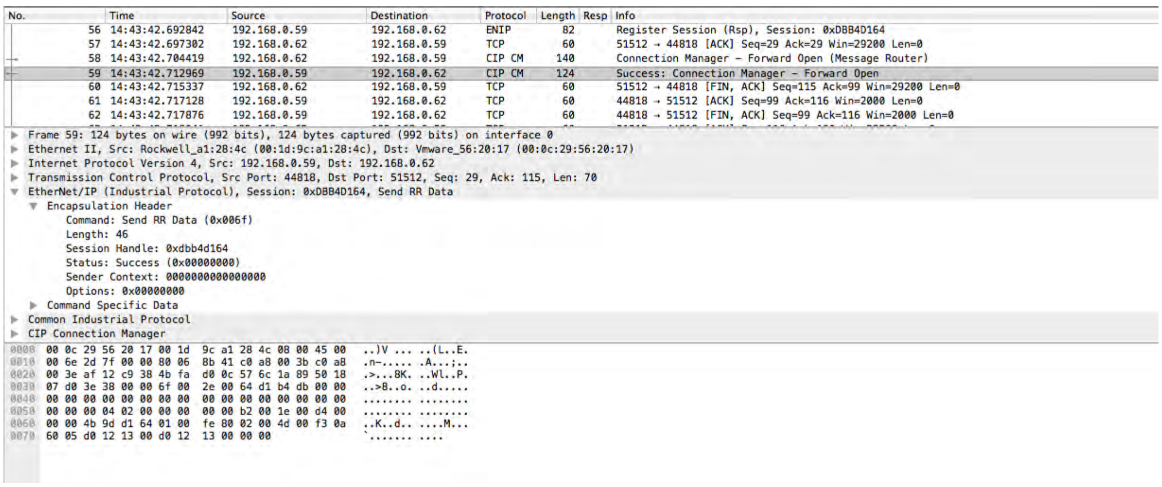


Figure 76. SendRRData Response over TCP (Fuzzed Options)

(5) T18 Results

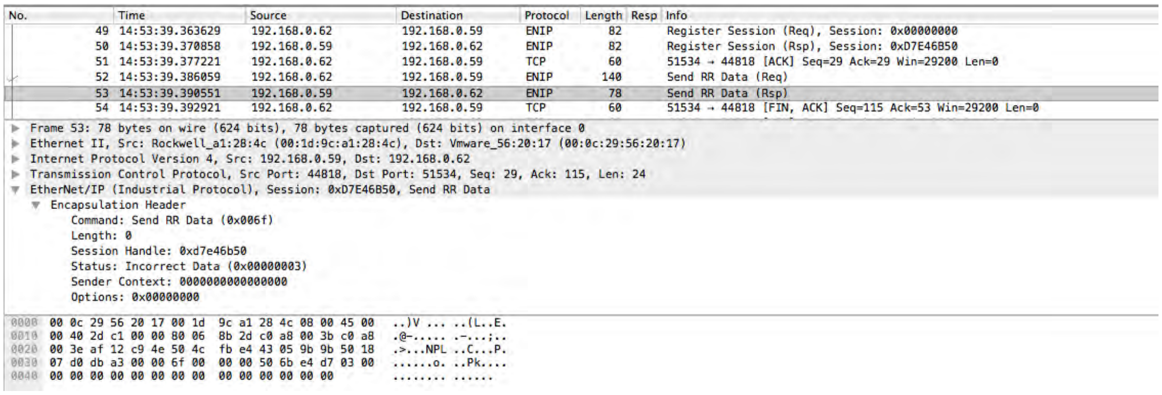


Figure 77. SendRRData Response over TCP (Fuzzed Interface Handle)

(6) T19 Results

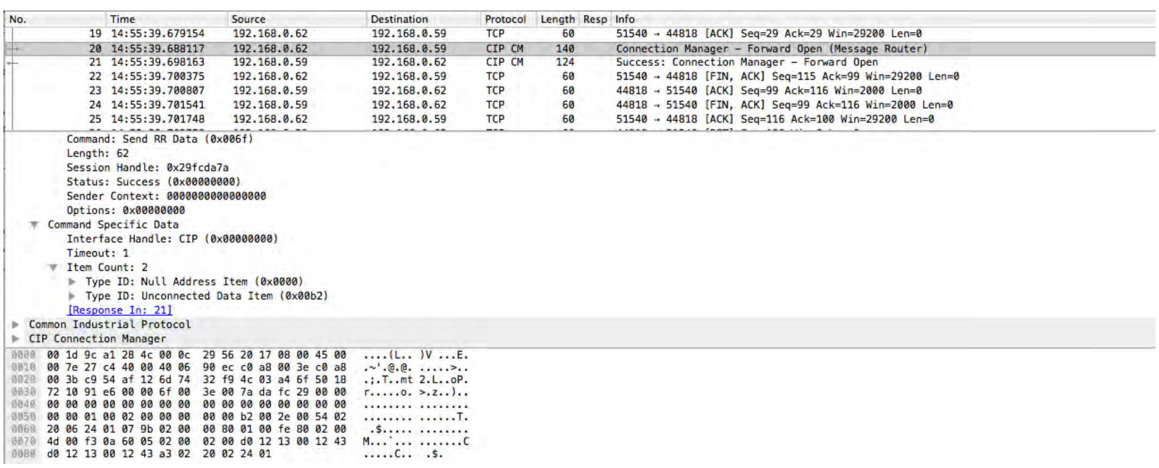


Figure 78. SendRRData Request over TCP (Fuzzed Timeout)

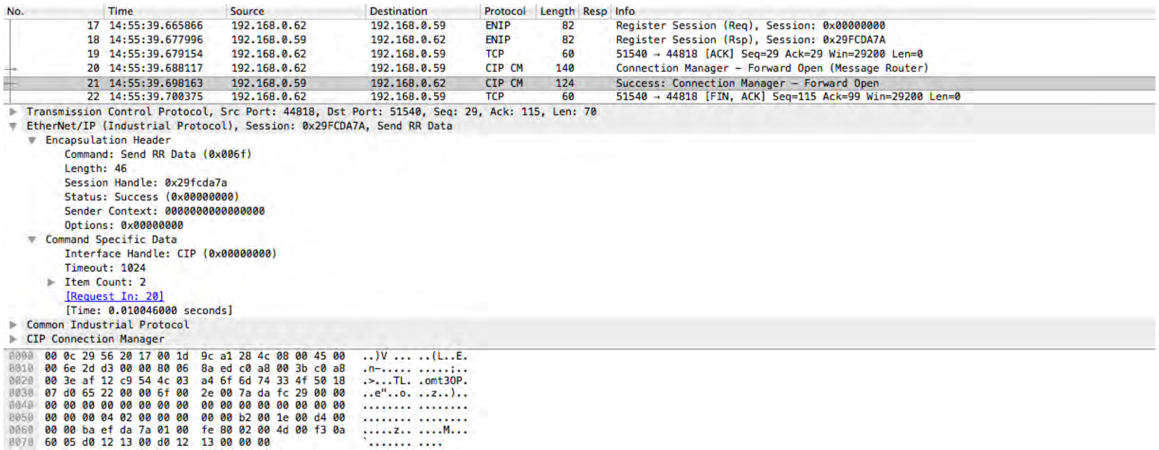


Figure 79. SendRRData Response over TCP (Fuzzed Timeout)

D. ENIP SENDUNITDATA TEST CASES

This section shows the results of the ENIP SendUnitData test cases.

(1) T20 Results

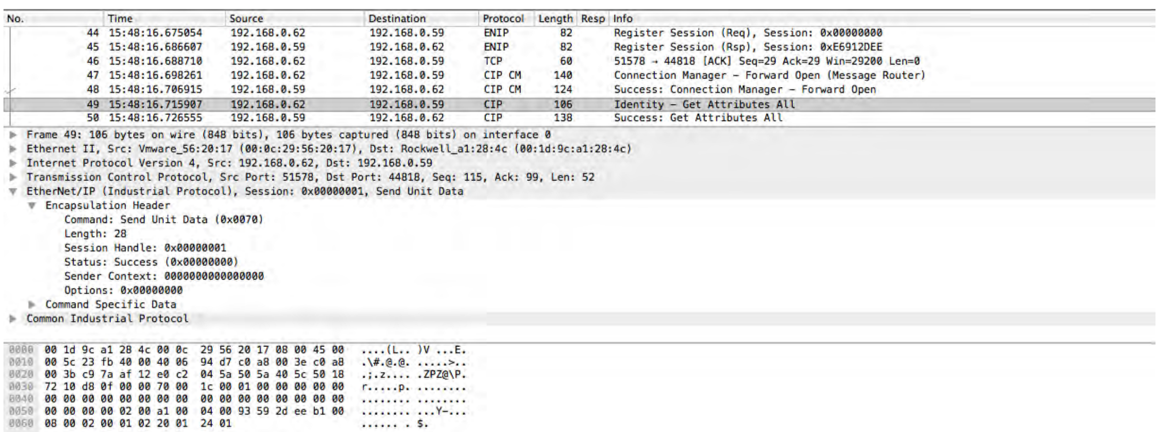


Figure 80. SendUnitData Request over TCP (Fuzzed Session Handle)

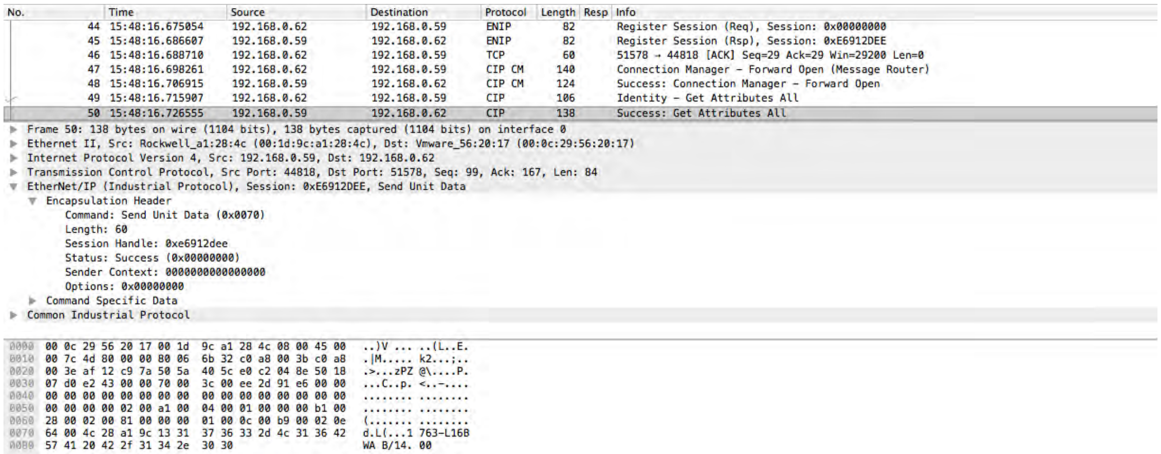


Figure 81. SendUnitData Response over TCP (Fuzzed Session Handle)

(2) T21 Results

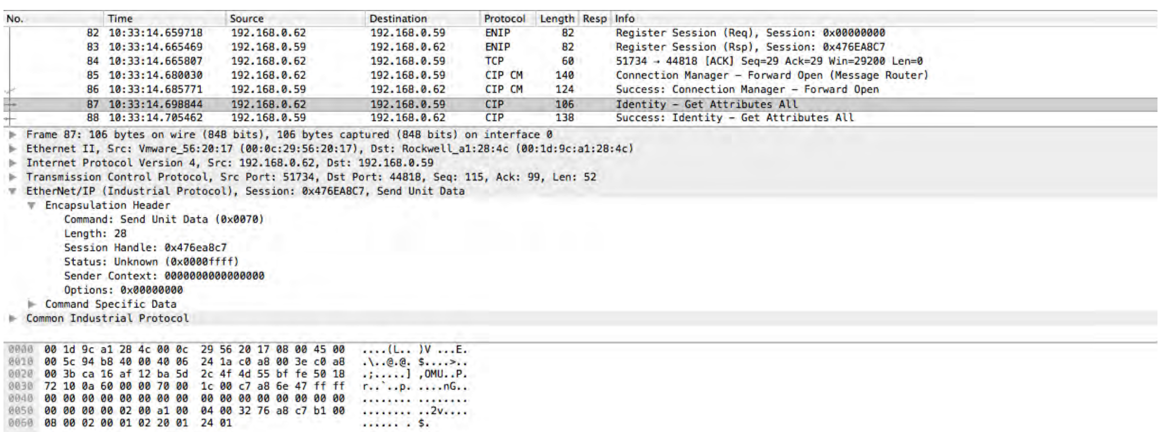


Figure 82. SendUnitData Request over TCP (Fuzzed Status: 0x0000FFFF)

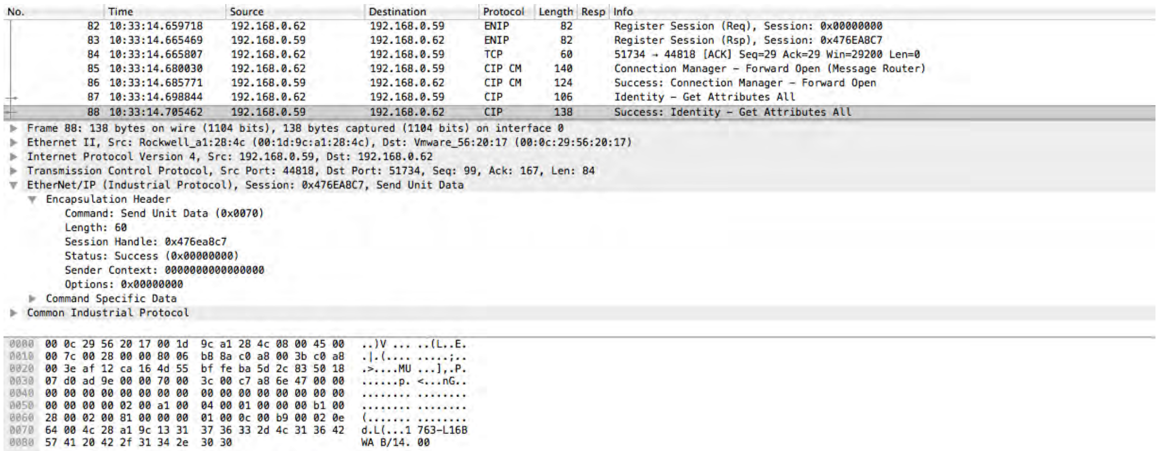


Figure 83. SendUnitData Response over TCP (Fuzzed Status: 0x0000FFFF)

(3) T22 Results

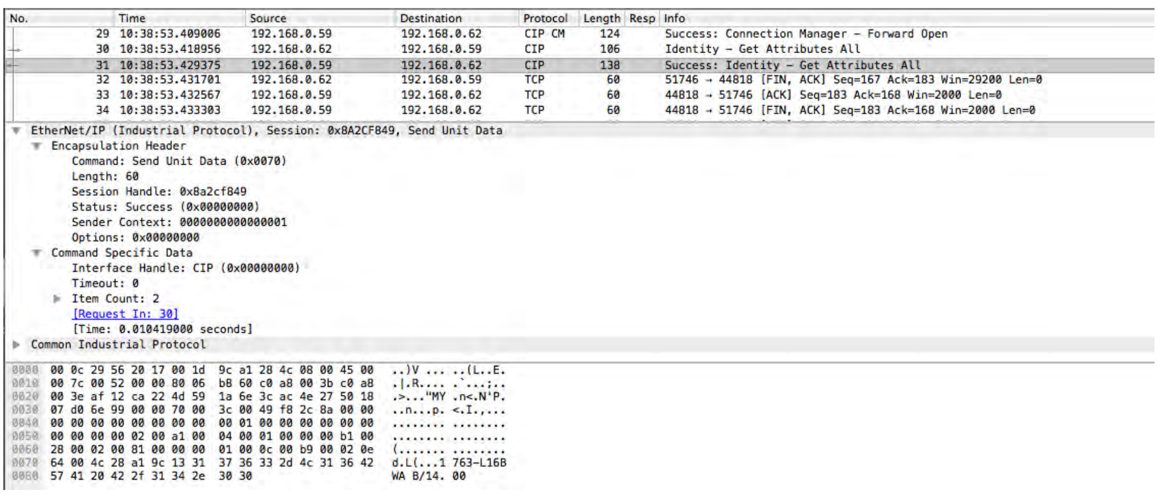


Figure 84. SendUnitData Response over TCP (Fuzzed Sender Context)

(4) T23 Results

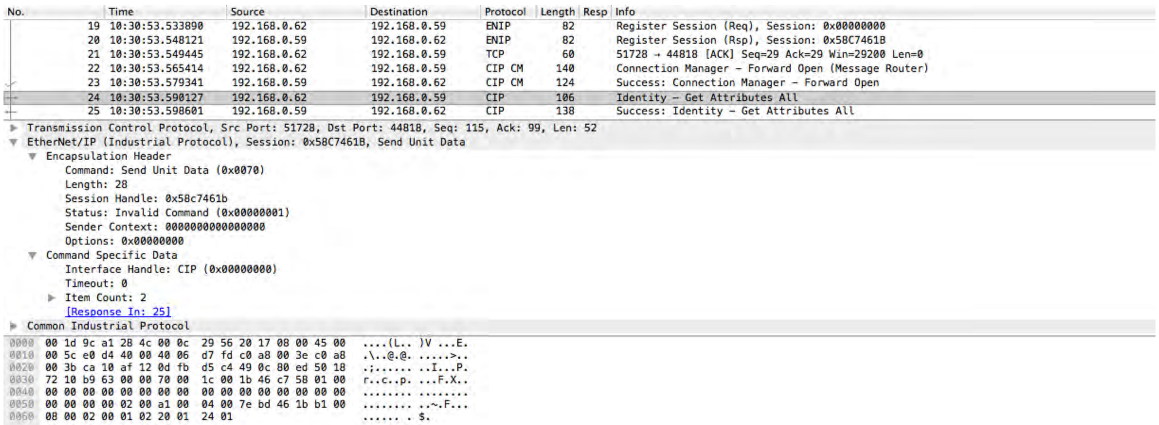


Figure 85. SendUnitData Request over TCP (Fuzzed Options)

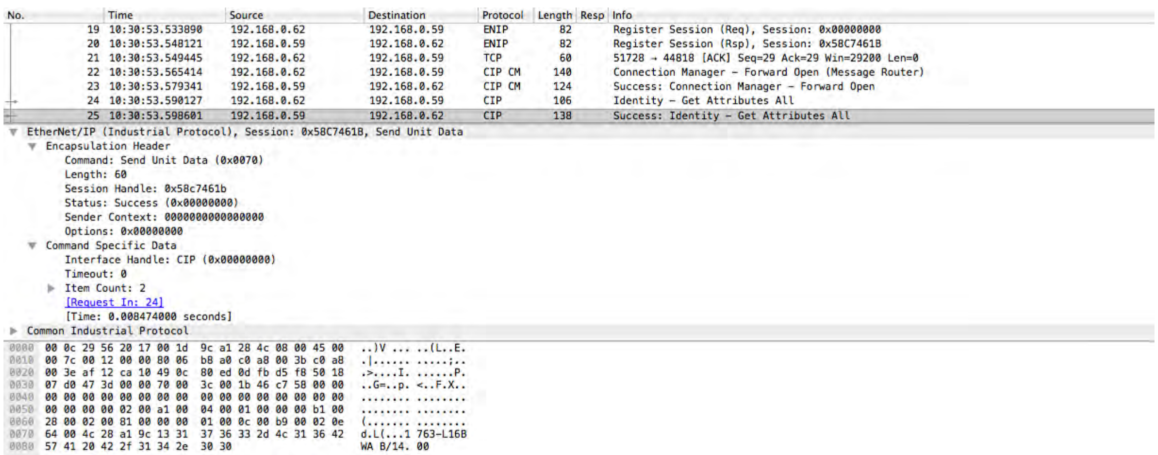


Figure 86. SendUnitData Response over TCP (Fuzzed Options)

(5) T24 Results

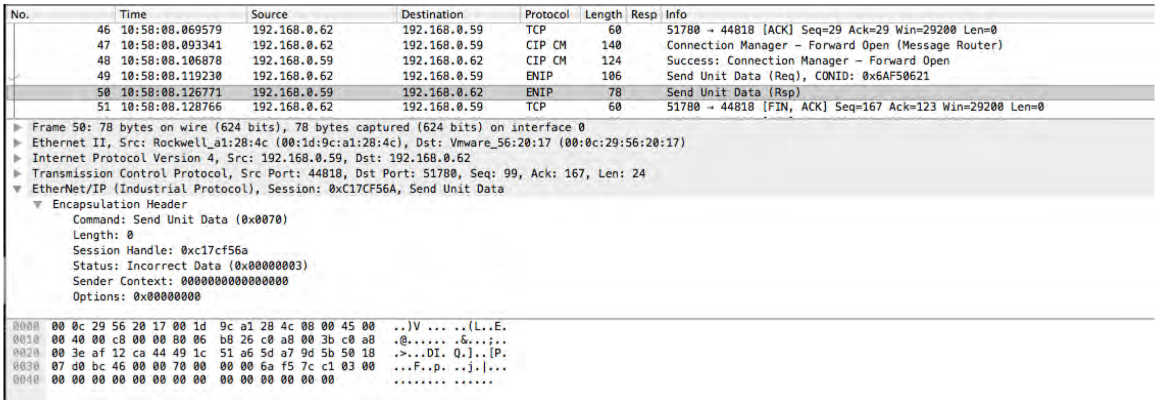


Figure 87. SendUnitData Response over TCP (Fuzzed Interface Handle)

(6) T25 Results

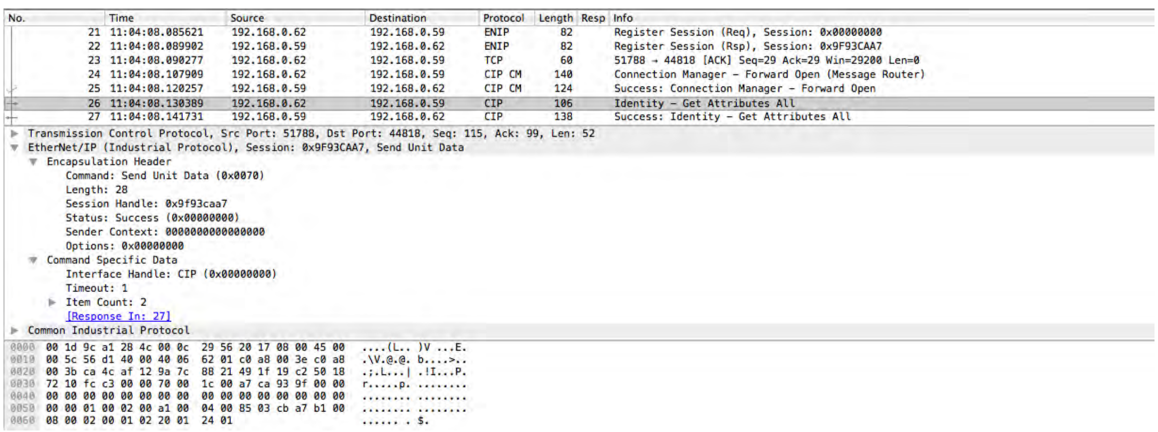


Figure 88. SendUnitData Request over TCP (Fuzzed Timeout)

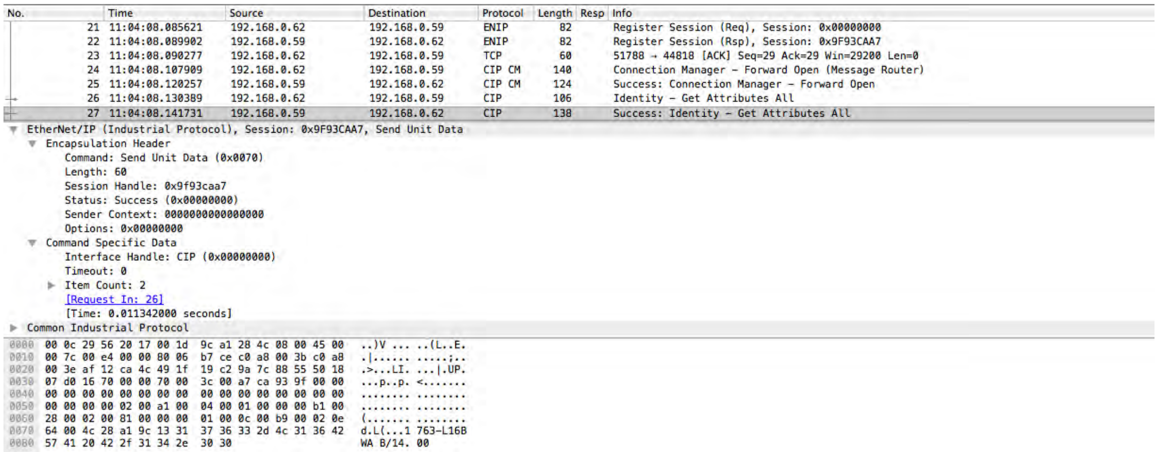


Figure 89. SendUnitData Response over TCP (Fuzzed Timeout)

E. ENIP RESERVED FOR LEGACY USE TEST CASES

This section shows the results of the ENIP Reserved for Legacy Use test cases.

(1) T26 Results

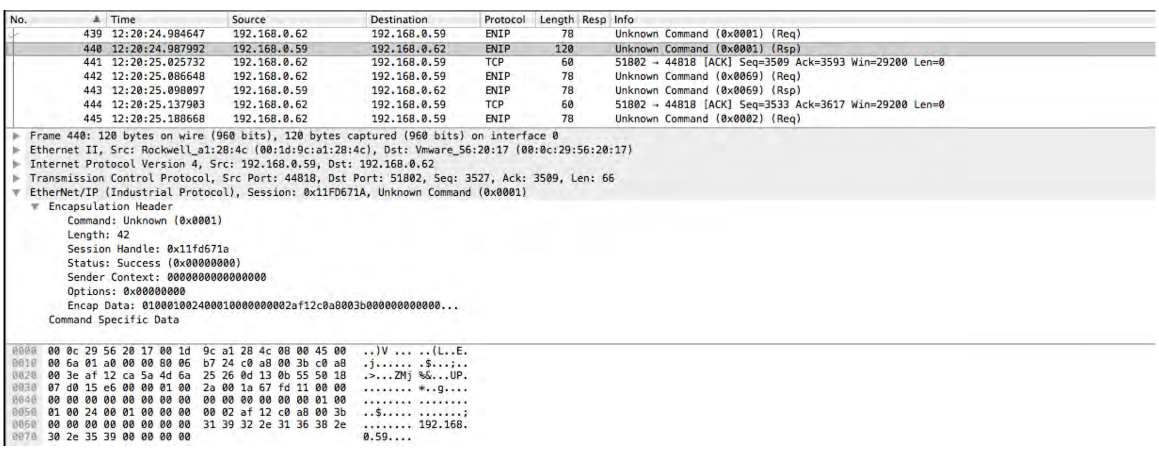


Figure 90. Reserved for Legacy Use Response over TCP
(Fuzzed Command Field)

(2) T27 Results

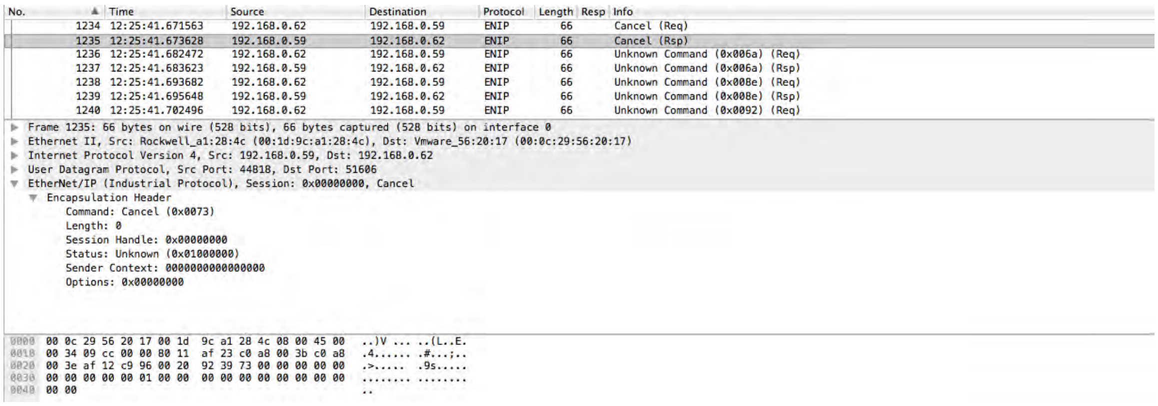


Figure 91. Reserved for Legacy Use Response over UDP
(Fuzzed Command Field)

F. ENIP RESERVED FOR FUTURE USE TEST CASES

This section shows the results of the ENIP Reserved for Future Use test cases.

(1) T28 Results

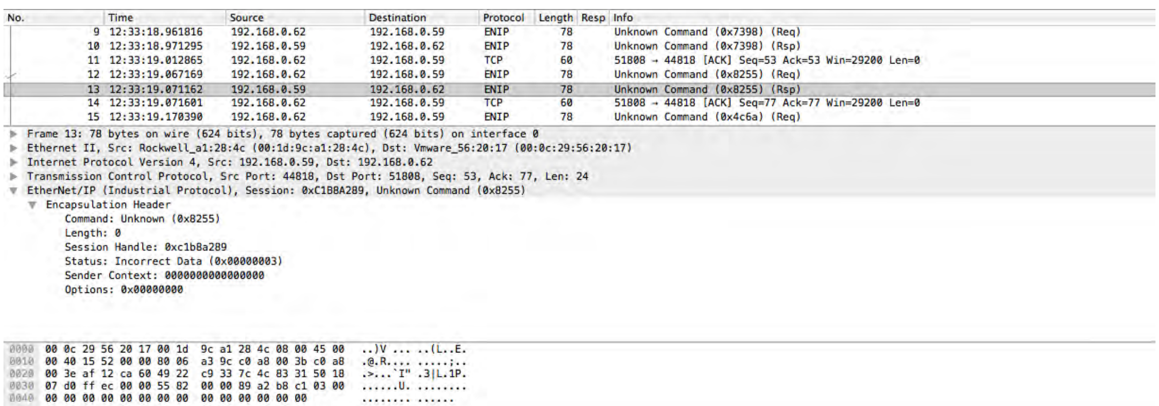


Figure 92. Reserved for Future Use Response over TCP
(Fuzzed Command Field)

(2) T29 Results

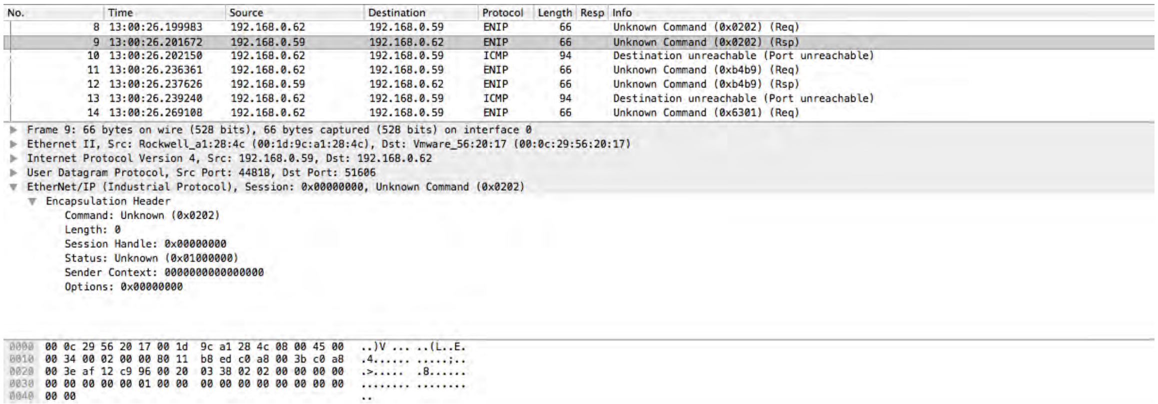


Figure 93. Reserved for Future Use Response over UDP (Fuzzed Command Field)

APPENDIX B. CIP COMMAND RESPONSES

The following Wireshark captures in Figures 94–118 illustrate test case responses for each command. For certain test cases, the corresponding request command sent to the SUT is also included to show how select fuzzed field inputs affect SUT responses. For descriptions of SUT responses, see Chapter V: Test Analysis.

A. CIP GET_ATTRIBUTES_ALL TEST CASES

This section shows the results of the CIP Get_Attributes_All test cases.

(1) T30 Results

The Get_Attributes_All request with a fuzzed Class field returns three types of responses. Figure 94 illustrates a successful CIP response. Figure 95 shows a “Service not supported” response. Figure 96 depicts a “Path destination unknown” response.

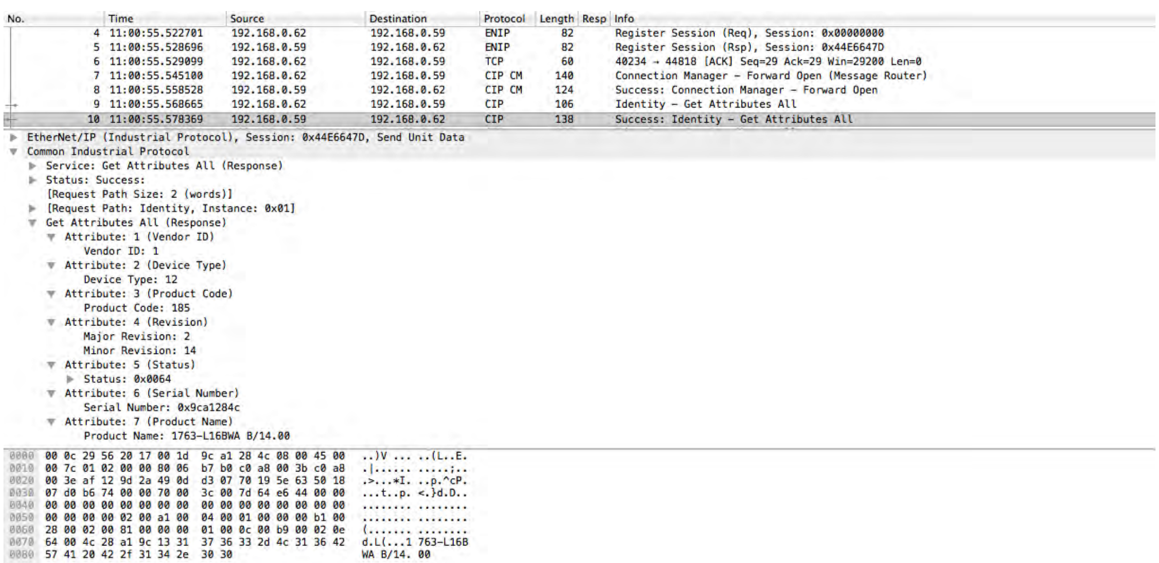


Figure 94. Get_Attributes_All Response over TCP (Class 0x01, Instance 0x01)

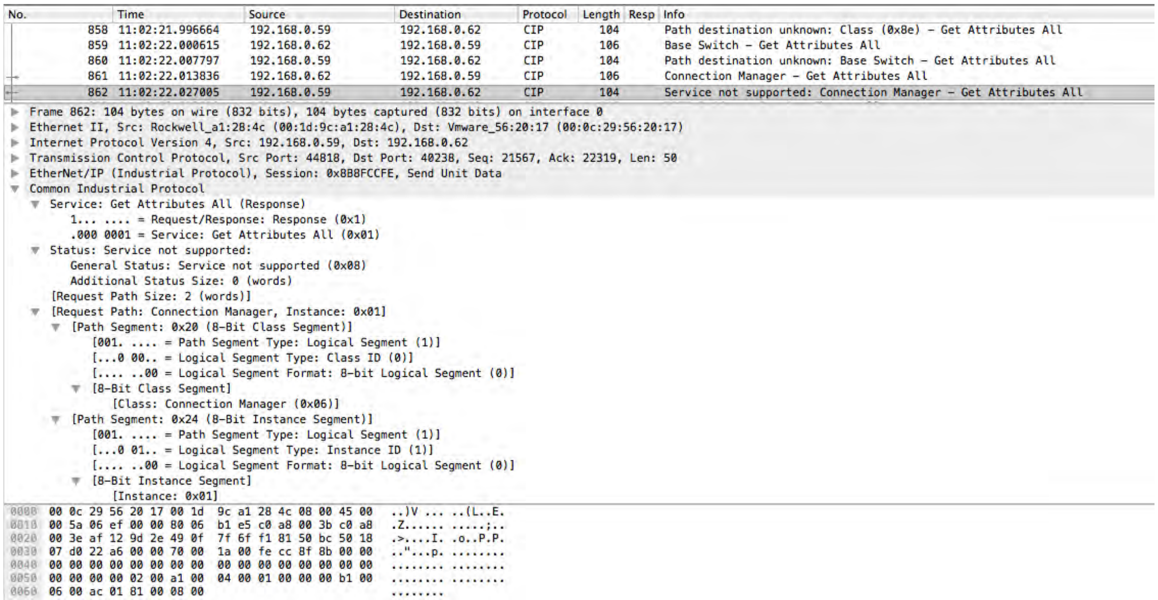


Figure 95. Get_Attributes_All “Service Not Supported” Response over TCP (Class 0x06, Instance 0x01)

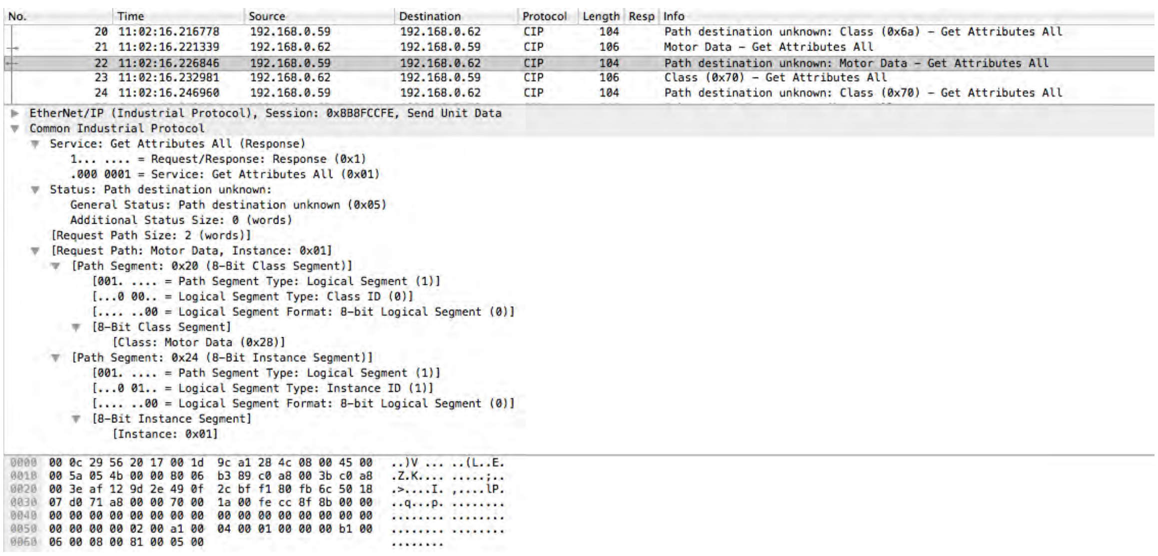


Figure 96. Get_Attributes_All “Path Destination Unknown” Response over TCP (Class 0x28, Instance 0x01)

(2) T31 Results

Figure 97 illustrates a “Path destination unknown response. Figure 98 demonstrates the response for the Identity Class with Instance 0x00.

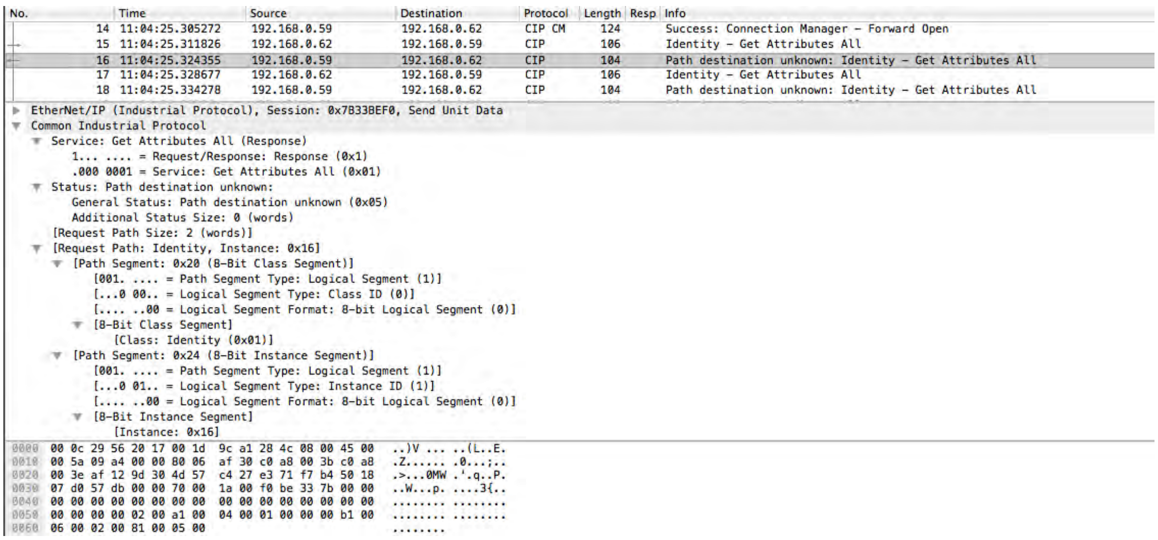


Figure 97. Get_Attributes_All “Path Destination Unknown” Response over TCP (Class 0x01, Instance 0x16)

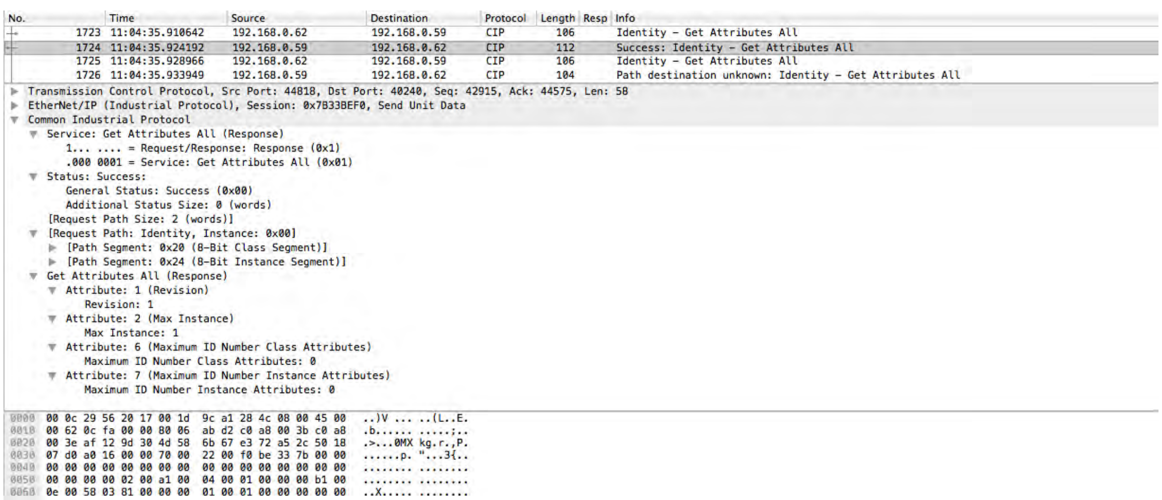


Figure 98. Get_Attributes_All Response over TCP (Class 0x01, Instance 0x00)

B. CIP GET_ATTRIBUTE_LIST TEST CASES

This section shows the results of the CIP Get_Attributes_List test cases.

(1) T32 Results

A Get_Attribute_List command with a fuzzed Class field returns two different responses. Figure 99 shows a General Status 0x08 “Service not supported” [15] response and Figure 100 illustrates a General Status 0x05 “Path destination unknown” [15] response.

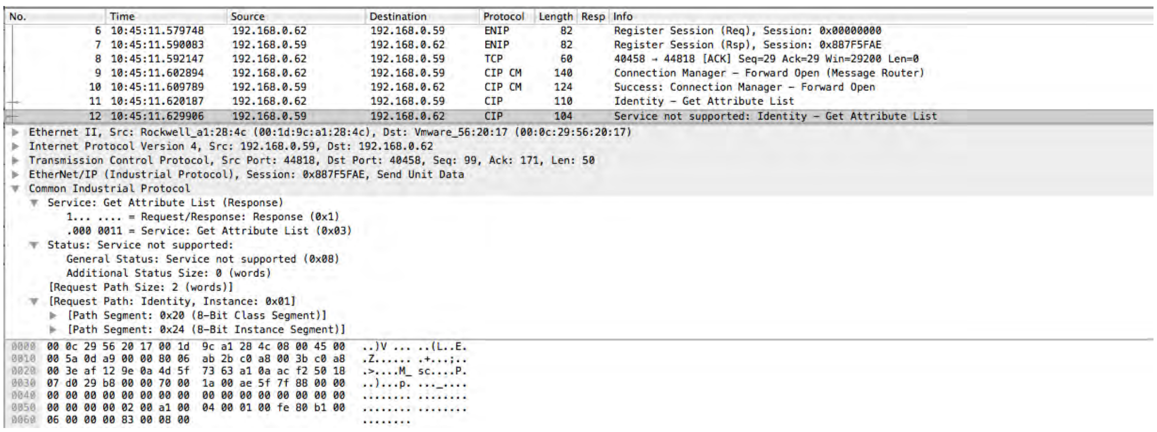


Figure 99. Get_Attribute_List Response over TCP (Class 0x01, Instance 0x01, Attribute 0x01)

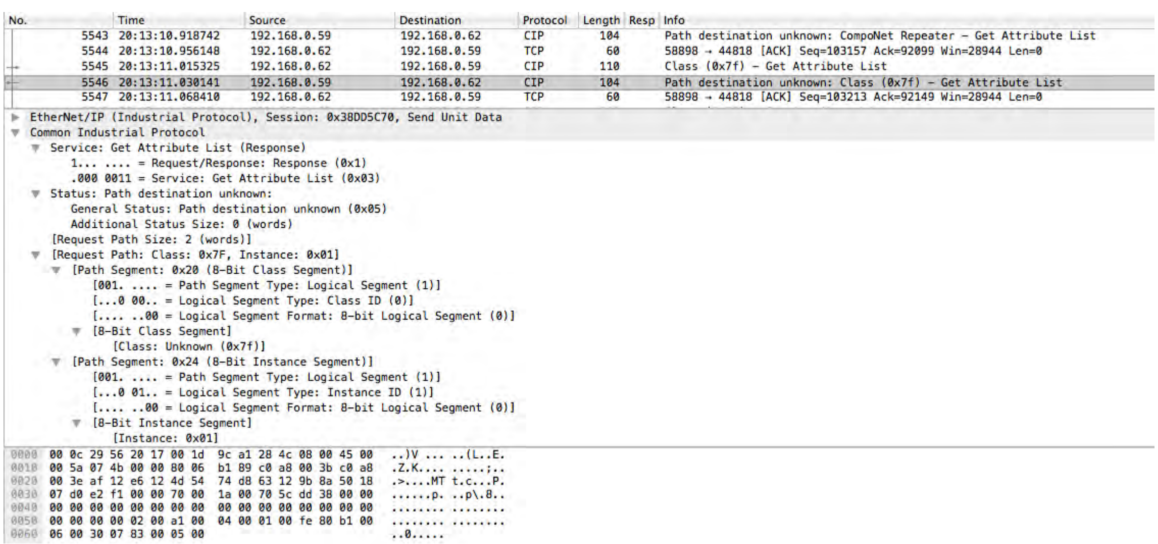


Figure 100. Get_Attribute_List Response over TCP (Class 0x7F, Instance 0x01, Attribute 0x01)

(2) T33 Results

The Get_Attribute_List with fuzzed Instance field requests return two different responses: General Status 0x08 “Service not supported” [15] responses (Figure 101) and General Status 0x05 “Path destination unknown” [15] responses (Figure 102).

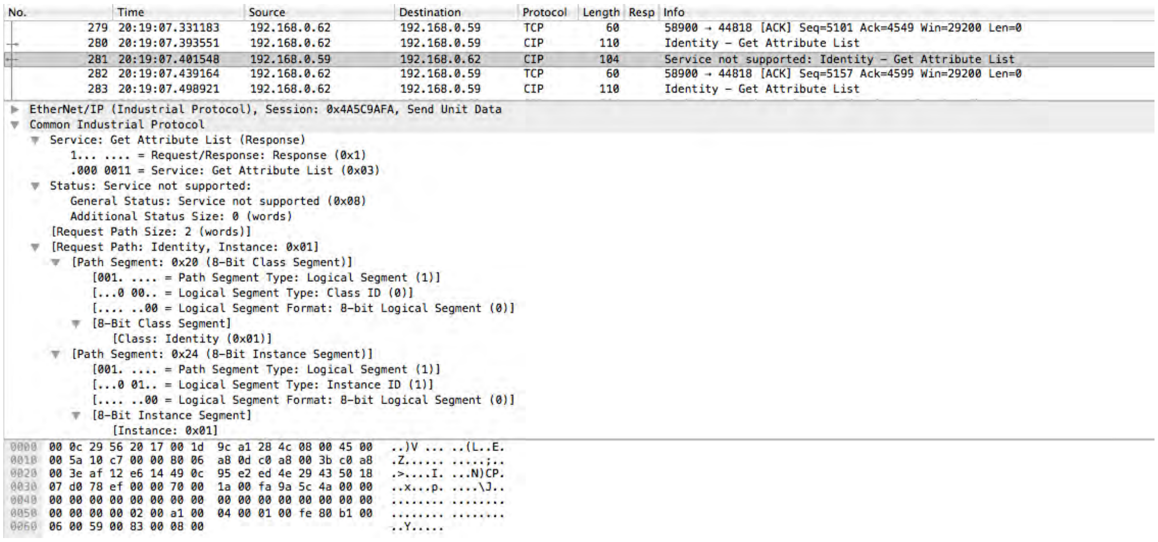


Figure 101. Get_Attribute_List “Service Not Supported” Response over TCP (Class 0x01, Instance 0x01, Attribute 0x01)

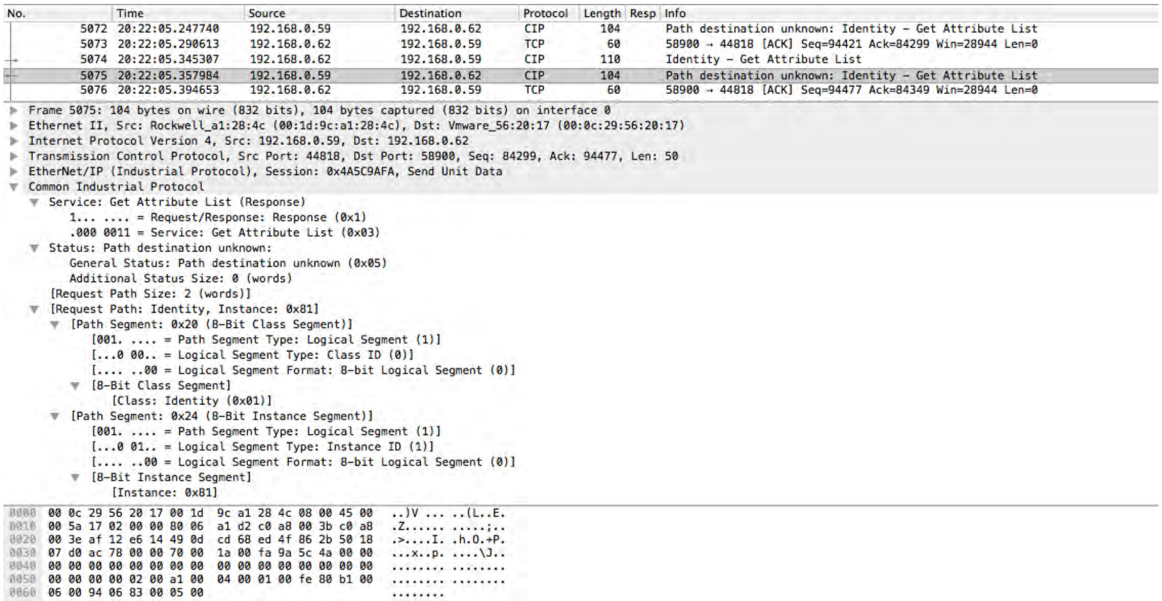


Figure 102. Get_Attribute_List “Path Destination Unknown” Response over TCP (Class 0x01, Instance 0x01, Attribute 0x01)

(3) T34 Results

The Get_Attribute_List command with a fuzzed Attribute field returns General Status 0x08 “Service not supported” [15] responses as shown in Figure 103.

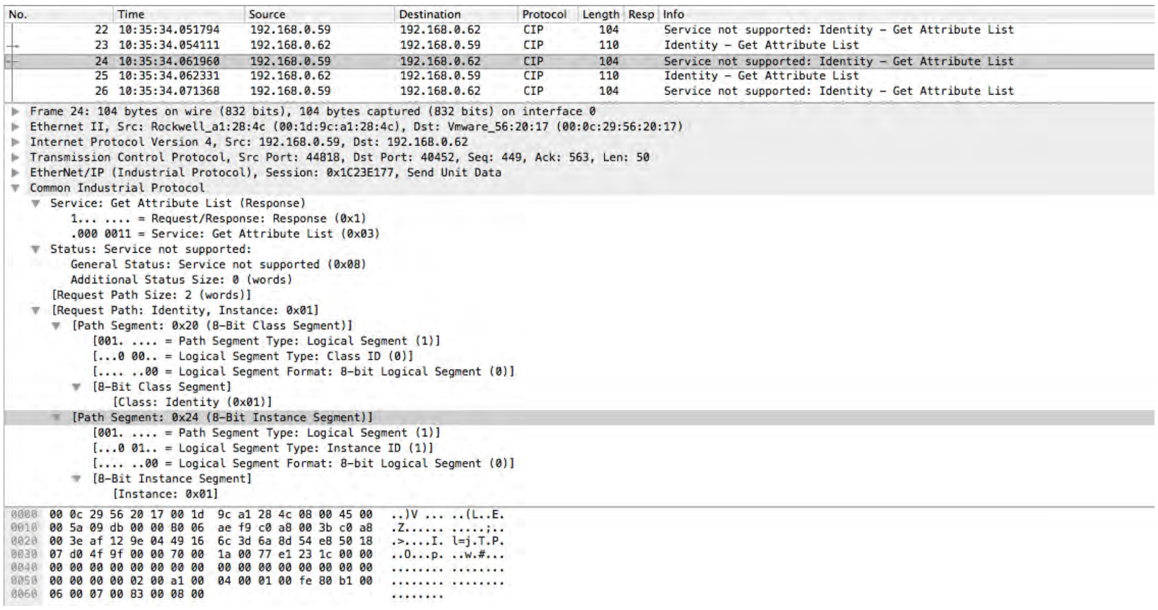


Figure 103. Get_Attribute_List “Service Not Supported” Response over TCP (Class 0x01, Instance 0x01, Attribute 200)

(4) T35 Results

A Get_Attribute_List request with the Attribute_count set to 223 is illustrated in Figure 104. Figure 105 shows the SUT response. Get_Attribute_List commands with the Attribute_count field exceeding 223 (Figure 106) receive a TCP ACK response (Figure 107).

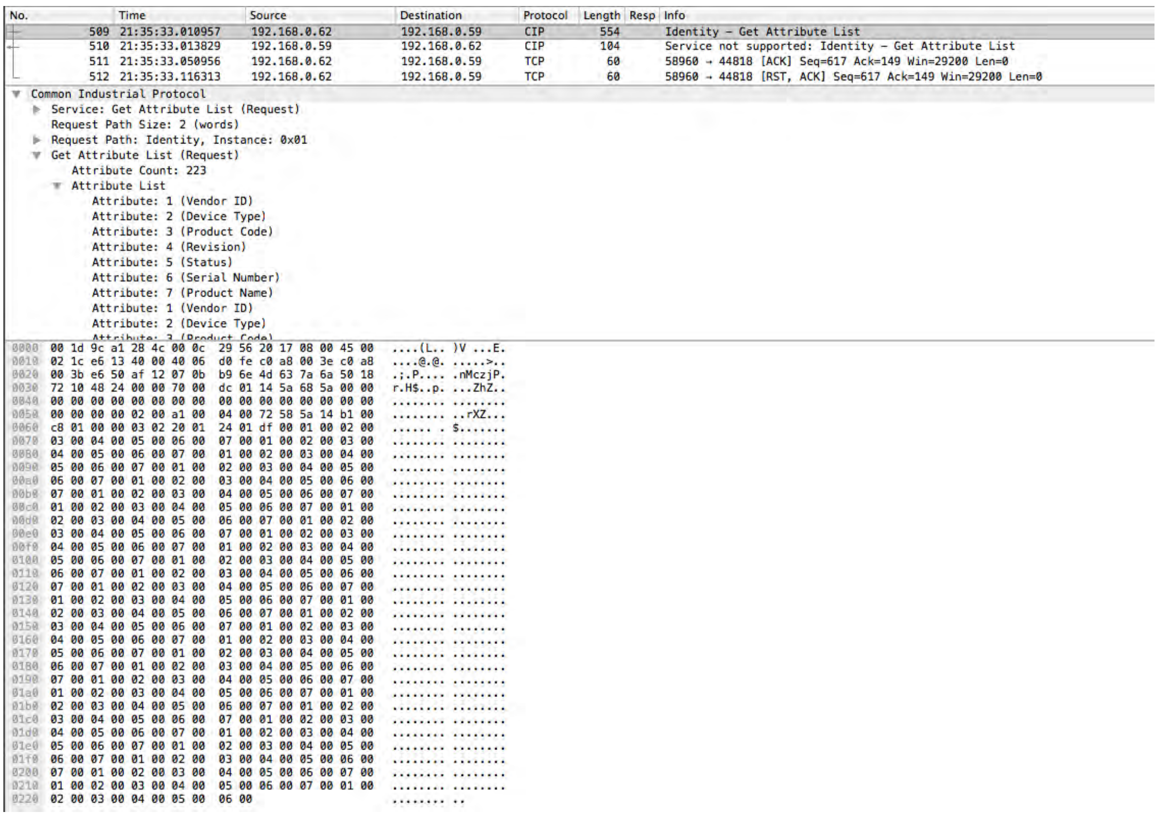


Figure 104. Get_Attribute_List Request over TCP (Attribute_count: 223)

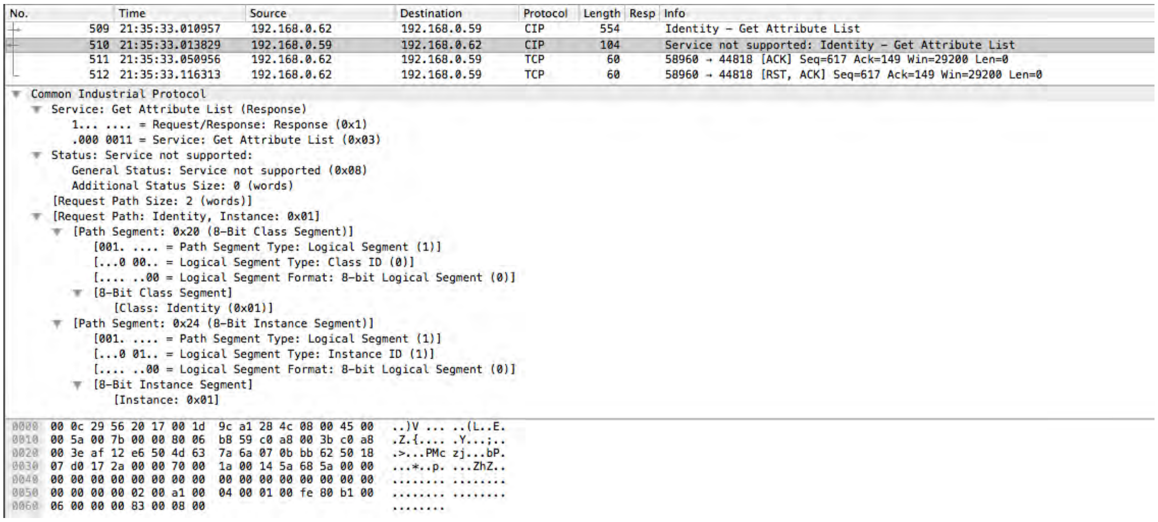


Figure 105. Get_Attribute_List Response over TCP (Attribute_count: 223)

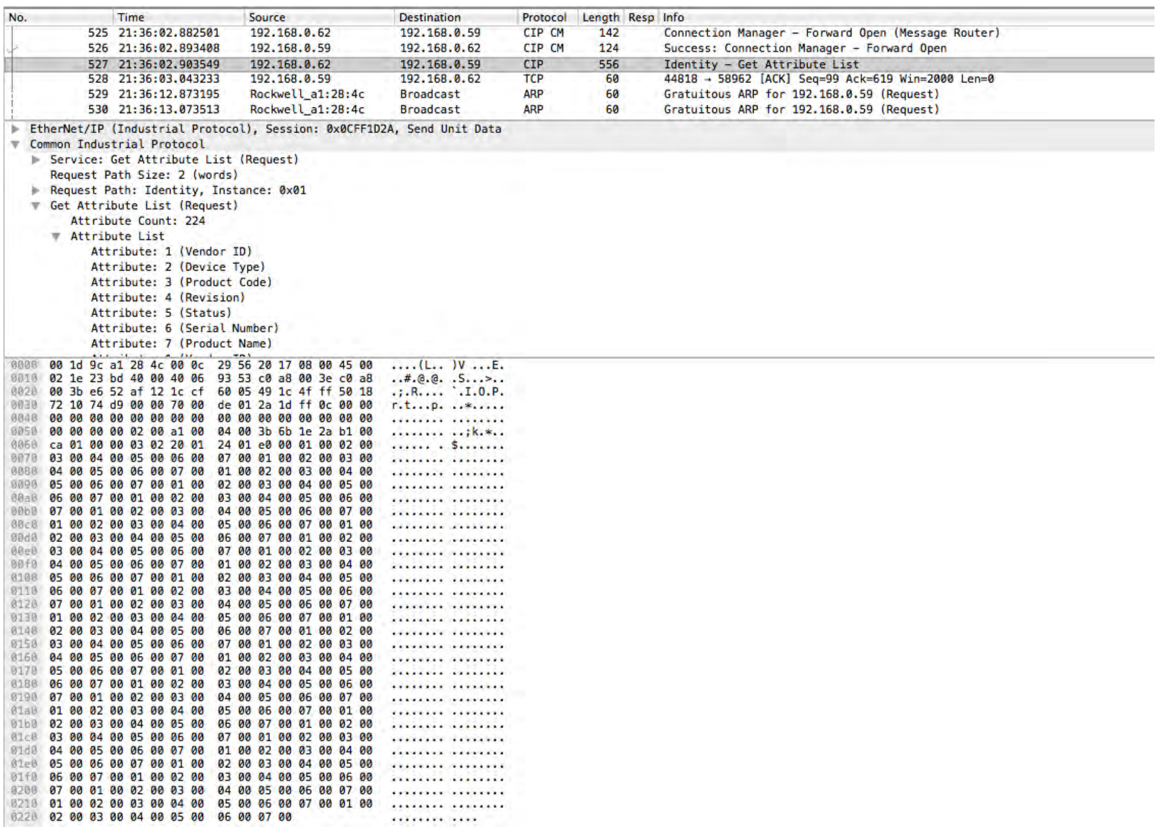


Figure 106. Get_Attribute_List Request over TCP (Attribute_count: 224)

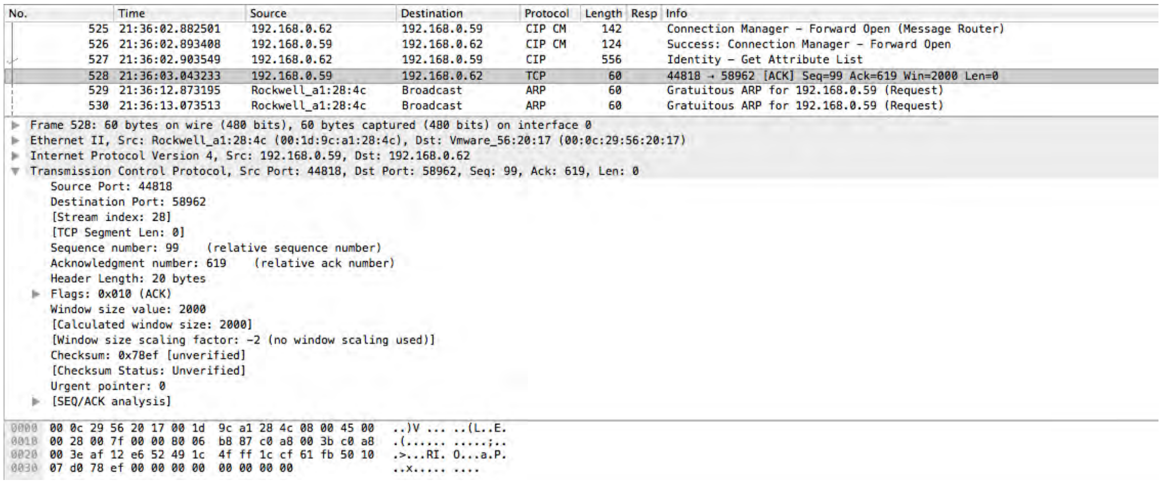


Figure 107. Get_Attribute_List Response over TCP (Attribute_count: 224)

C. CIP GET_ATTRIBUTE_SINGLE TEST CASES

This section shows the results of the CIP Get_Attributes_Single test cases.

(1) T36 Results

Get_Attribute_Single with a fuzzed Class field returns either a “Service not supported” response (Figure 108) or a “Path destination unknown response” (Figure 109).

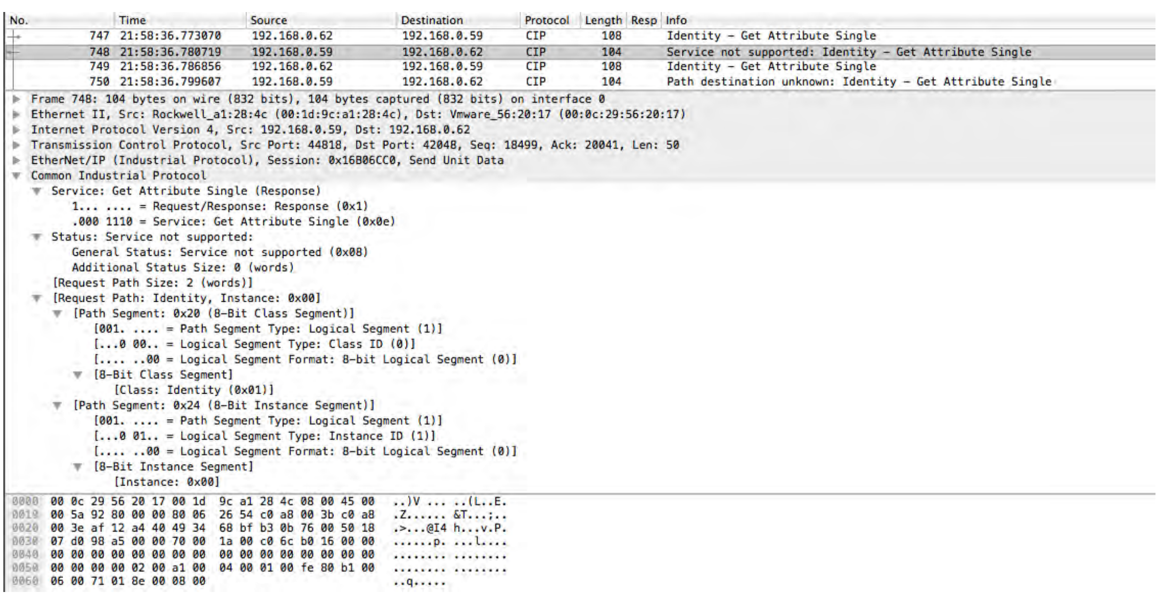


Figure 108. Get_Attribute_Single “Service Not Supported” Response over TCP

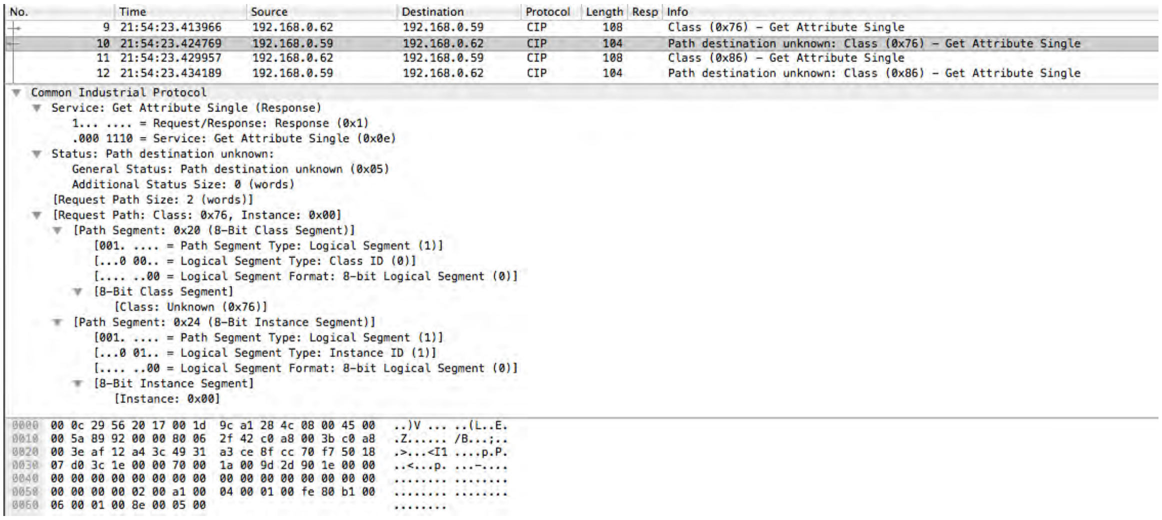


Figure 109. Get_Attribute_Single "Path Destination Unknown" Response over TCP

(2) T37 Results

The Get_Attribute_Single command returns an "Attribute not supported" response when the Instance field is set to 0x00 and Class and Attribute fields are 0x01 (Figure 110). When the Instance field is 0x01, with the same Class and Attribute fields, the SUT returns a "Service not supported" message (Figure 111). All other Instance fields with the Class and Attribute fields set to 0x01 return "Path destination unknown" (Figure 112).

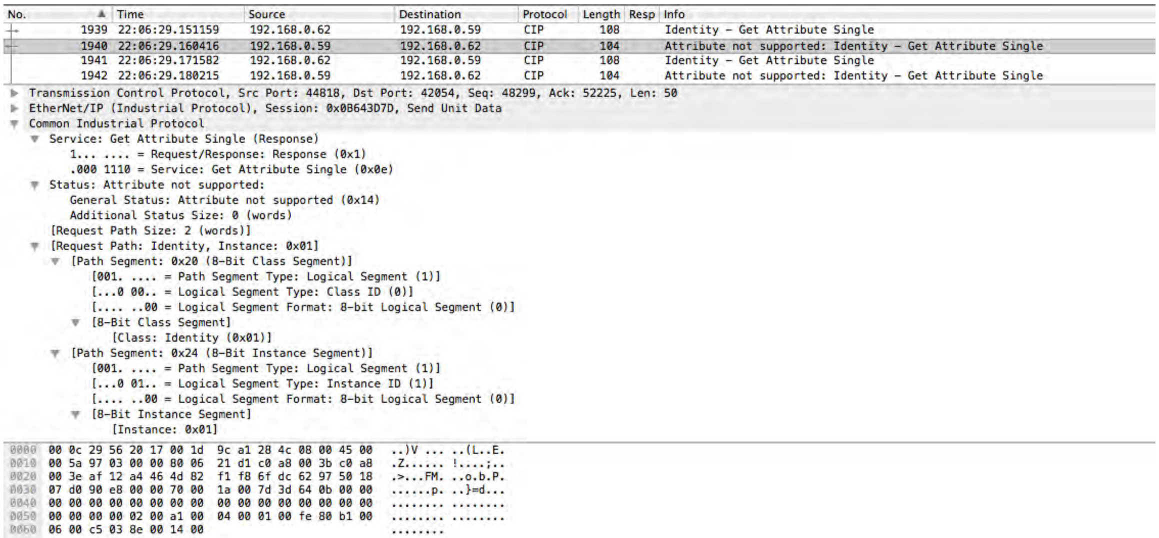


Figure 110. Get_Attribute_Single “Attribute Not Supported” Response over TCP

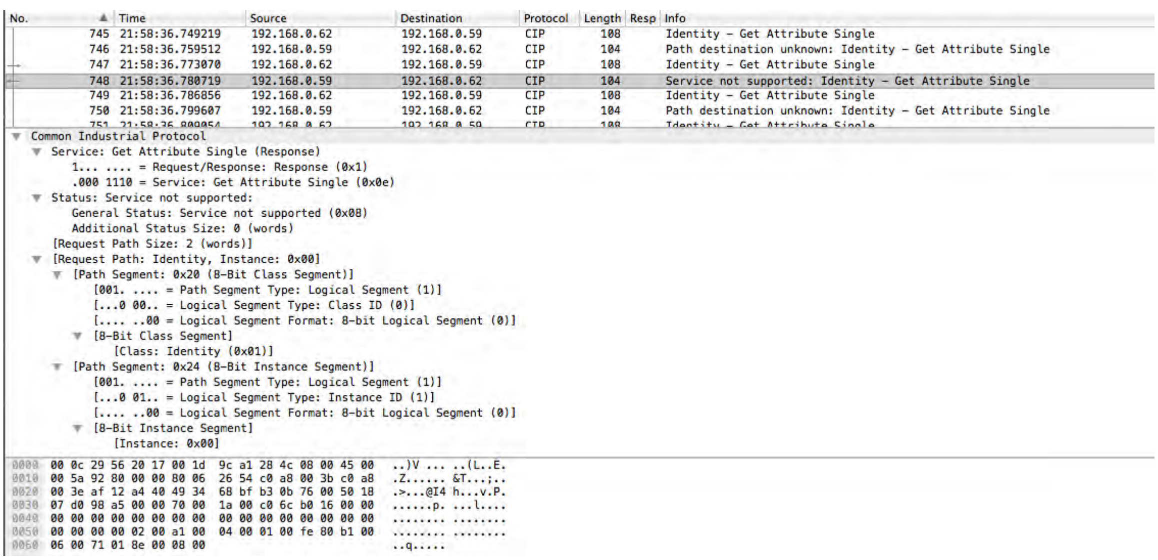


Figure 111. Get_Attribute_Single “Service Not Supported” Response over TCP

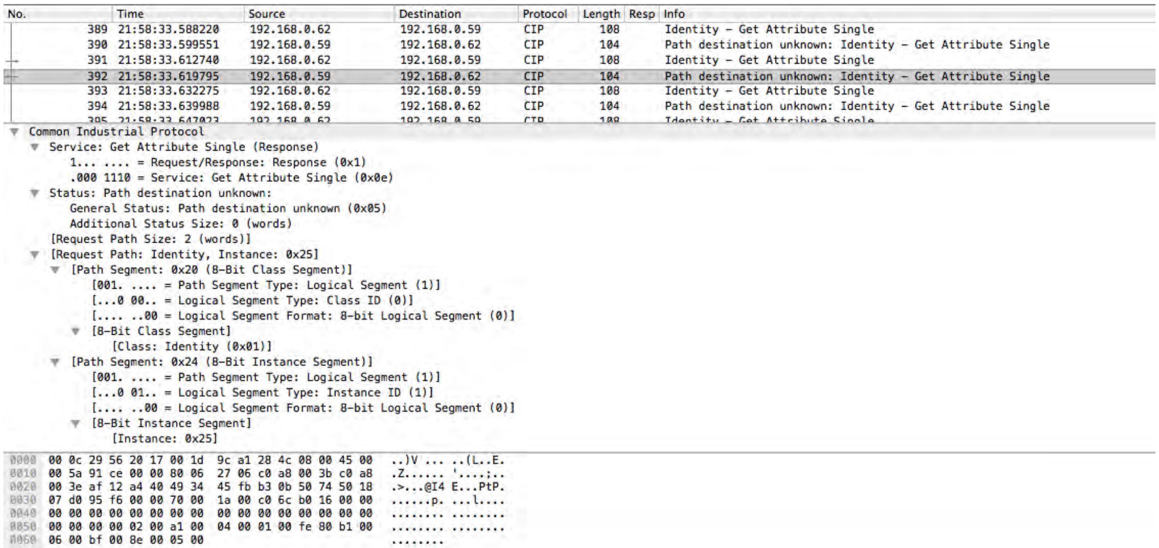


Figure 112. Get_Attribute_Single “Path Destination Unknown” Response over TCP

(3) T38 Results

The MicroLogix responds to all Get_Attribute_Single requests with a fuzzed Attribute field and the Class and Instance fields set to 0x00 with an “Attribute not supported” message (Figure 113).

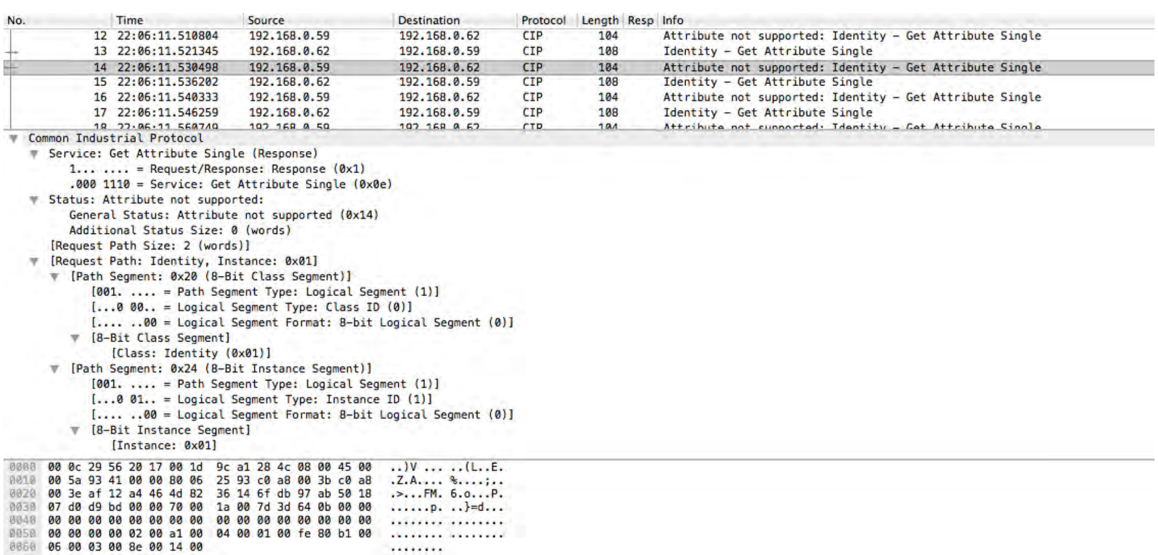


Figure 113. Get_Attribute_Single “Attribute Not Supported” Response over TCP

D. CIP FIND_NEXT_OBJECT_INSTANCE TEST CASES

This section shows the results of the CIP Find_Next_Object_Instance test cases.

(1) T39 Results

The CIP Find_Next_Object_Instance command with a fuzzed Class field returns "Service not supported" for six Class field inputs, as illustrated by Figure 114. All other fuzzed Classes returned "Path destination unknown" responses (Figure 115).

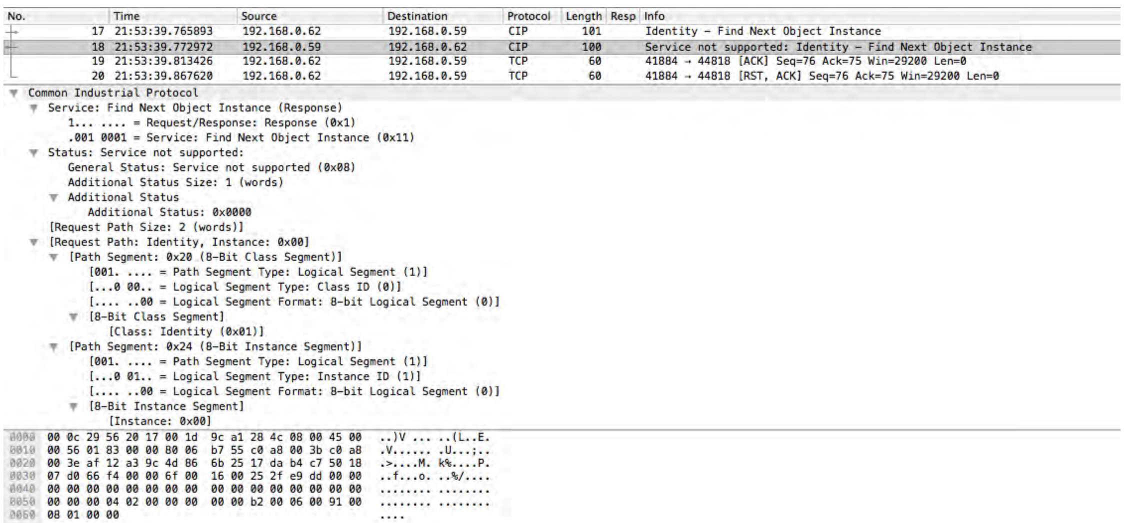


Figure 114. Find_Next_Object_Instance "Service Not Supported" Response over TCP

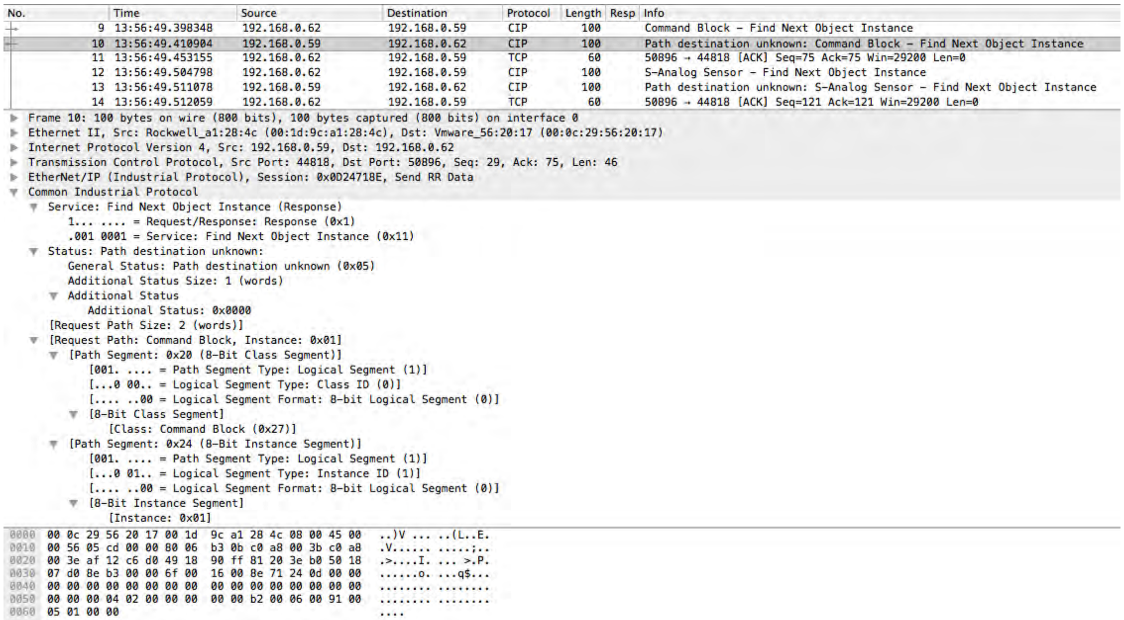


Figure 115. Find_Next_Object_Instance "Path Destination Unknown" Response over TCP

(2) T40 Results

When testing the Instance field, Class is set to 0x01. Requests with Instance 0x00 and 0x01 return "Service not supported" responses (Figure 116). All other fuzzed Instance inputs return "Path destination unknown" messages (Figure 117).

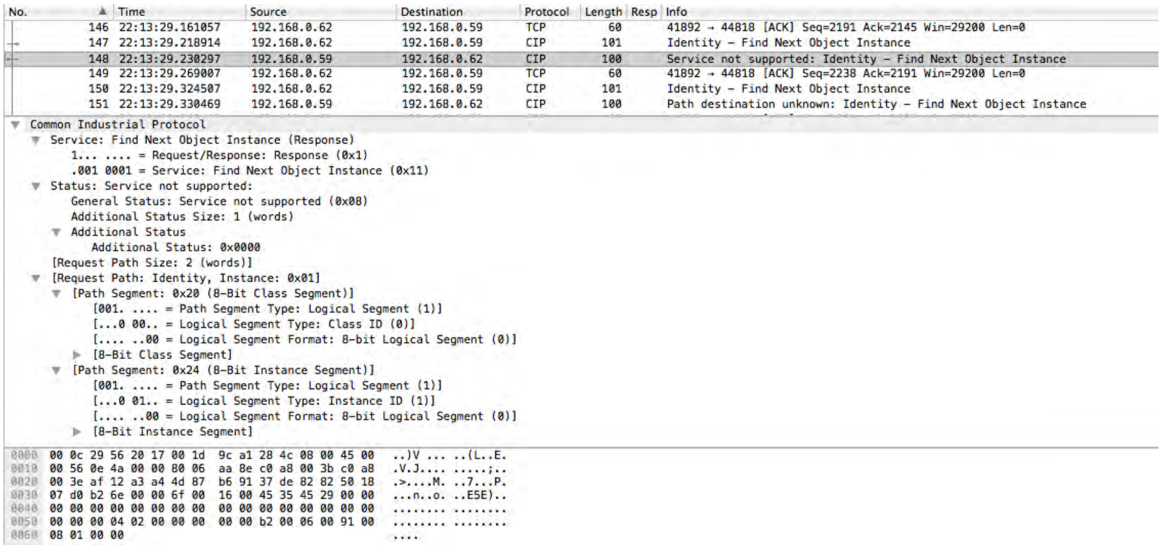


Figure 116. Find_Next_Object_Instance "Service Not Supported" Response over TCP

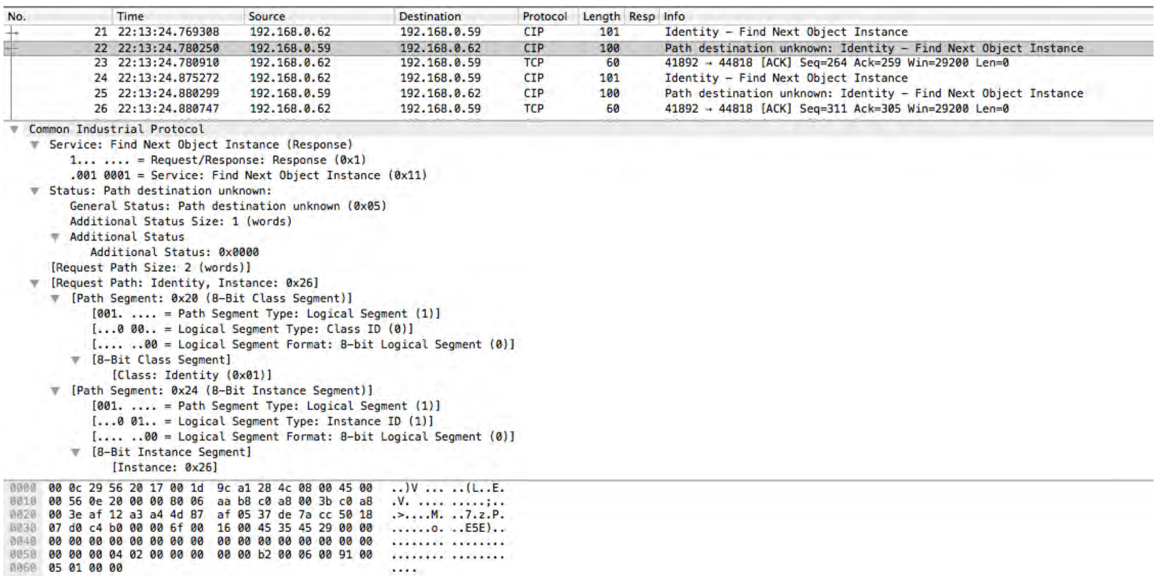


Figure 117. Find_Next_Object_Instance "Path Destination Unknown" Response over TCP

(3) T41 Results

The Maximum Returned Values field is tested with inputs between 0x00 and 0xFF. All requests return General Status 0x08 “Service not supported” [15] responses when the Class is set to 0x01 and Instance is set to 0x00 (Figure 118).

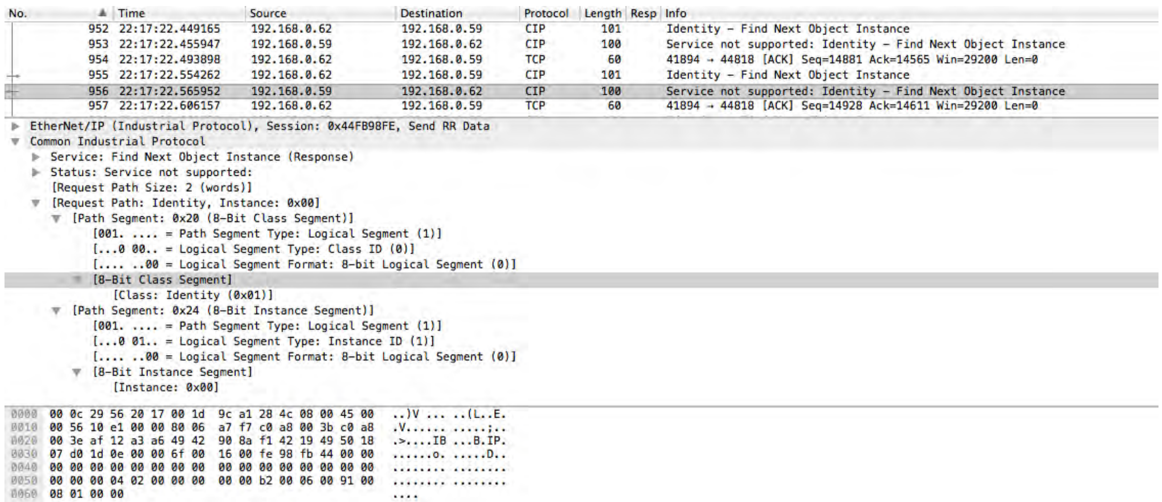


Figure 118. Find_Next_Object_Instance “Service Not Supported” Response over TCP

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. PCCC COMMAND RESPONSES

The following Wireshark captures in Figures 119 - 154 illustrate test case responses for each command. For descriptions of SUT responses, see Chapter V: Test Analysis.

A. PCCC ECHO TEST CASES

This section shows the results of the PCCC Echo test cases.

(1) T42 Results

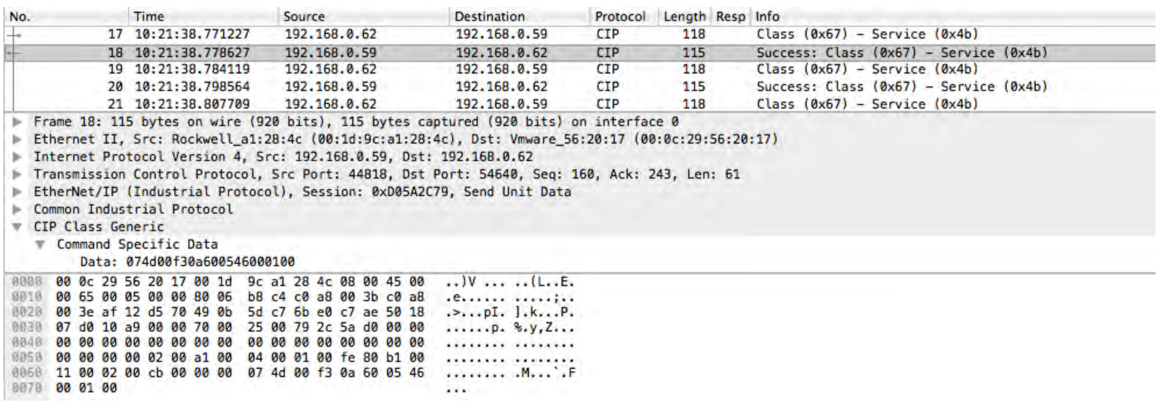


Figure 119. Echo Response over TCP (0 Bytes Attached)

(2) T43 Results

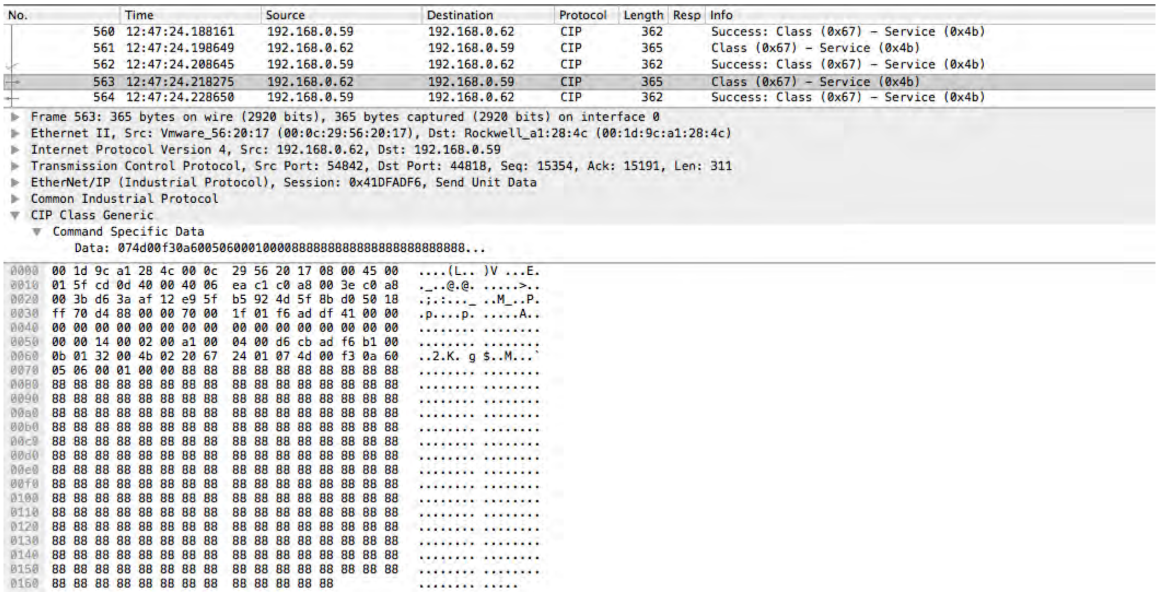


Figure 120. Echo Response over TCP (243 Bytes Attached)

(3) T44 Results

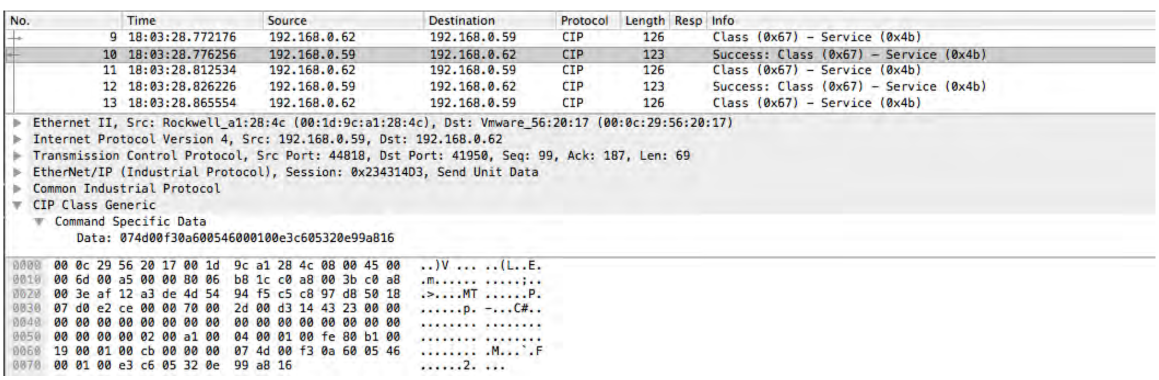


Figure 121. Echo Response over TCP (8 Bytes Fuzzed)

(4) T45 Results

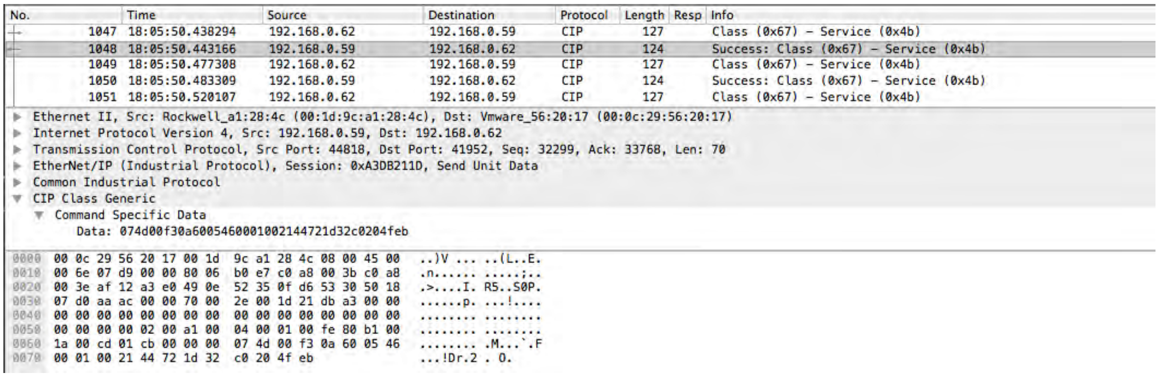


Figure 122. Echo Response over TCP (9 Bytes Fuzzed)

(5) T46 Results

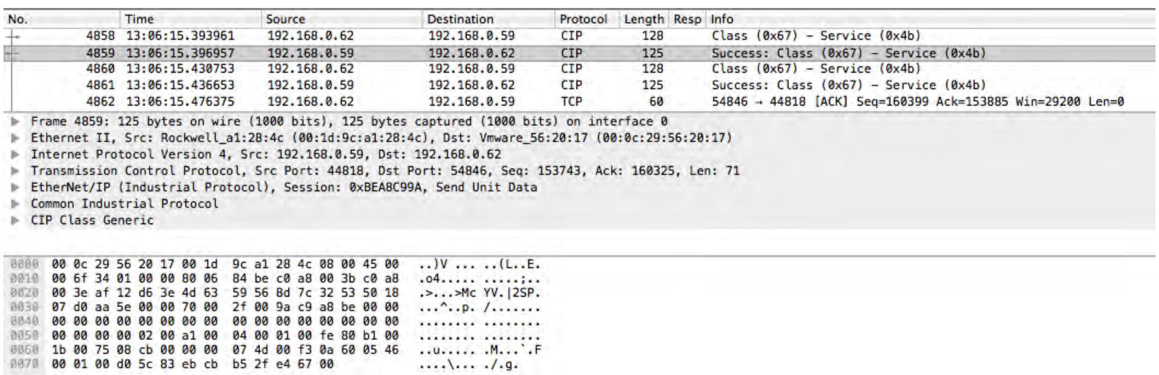


Figure 123. Echo Response over TCP (10 Bytes Fuzzed)

(6) T47 Results

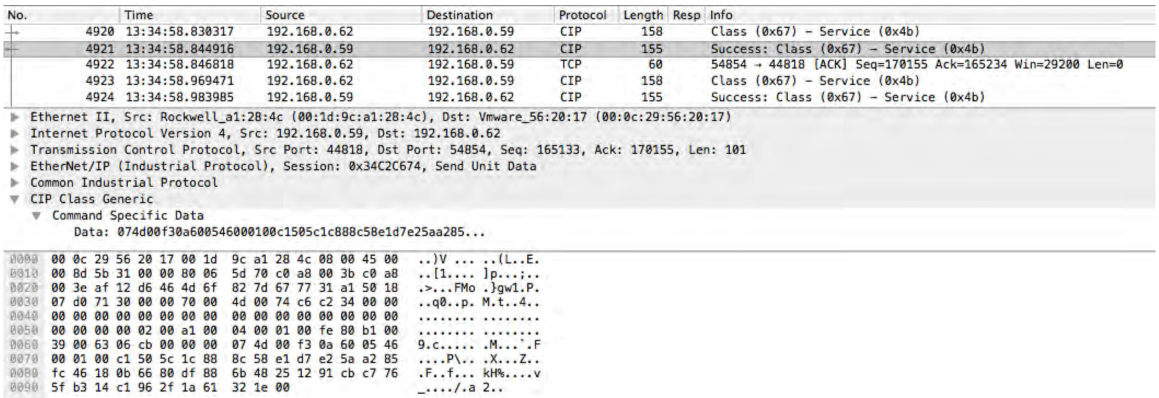


Figure 124. Echo Response over TCP (40 Bytes Fuzzed)

(7) T48 Results

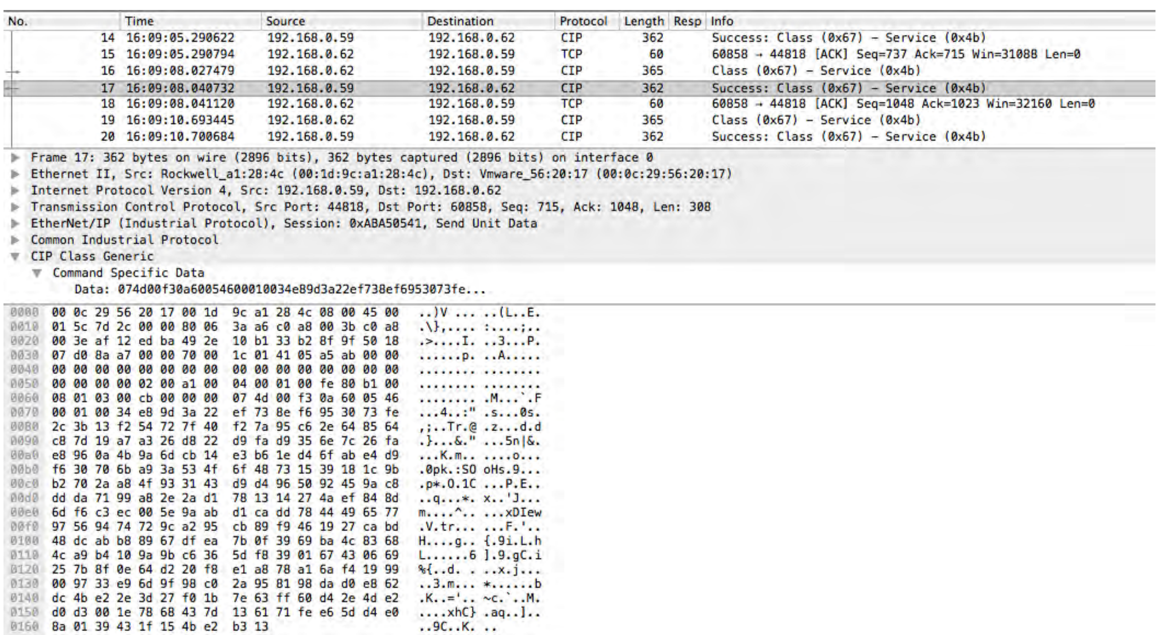


Figure 125. Echo Response over TCP (243 Bytes Fuzzed)

(8) T49 Results

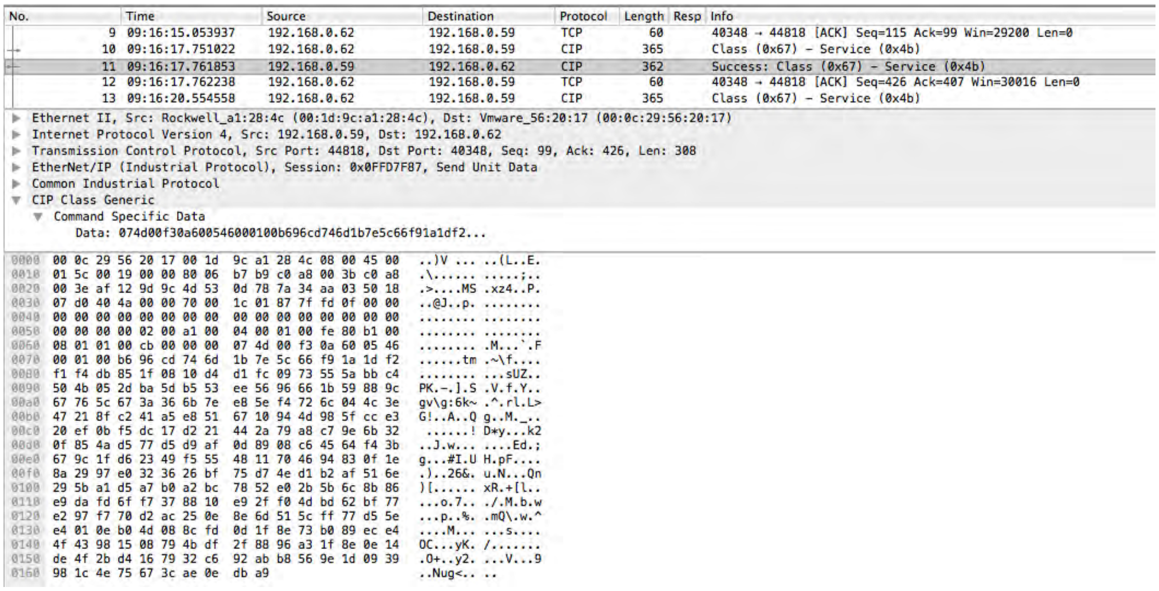


Figure 126. Echo Response over TCP (Maximum Number of Bytes without Error Message: 247 Bytes Fuzzed)

(9) T50 Results

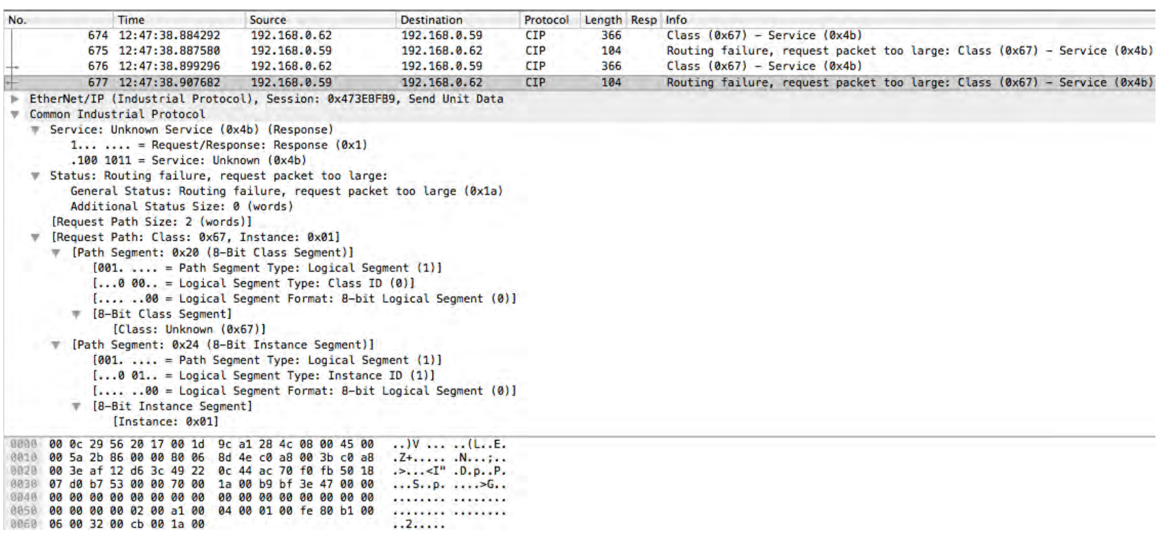


Figure 127. Echo Response over TCP (248 Bytes Fuzzed)

(10) T51 Results

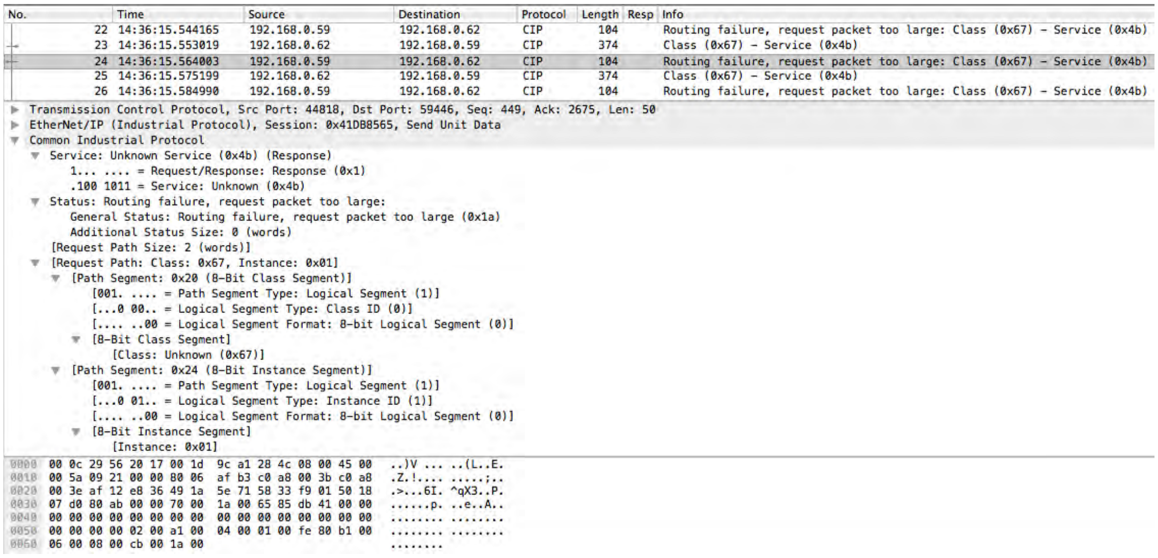


Figure 128. Echo Response over TCP (256 Bytes Fuzzed)

B. PCCC PROTECTED TYPED FILE READ TEST CASES

This section shows the results of the PCCC Protected Typed File Read test cases.

(1) T52 Results

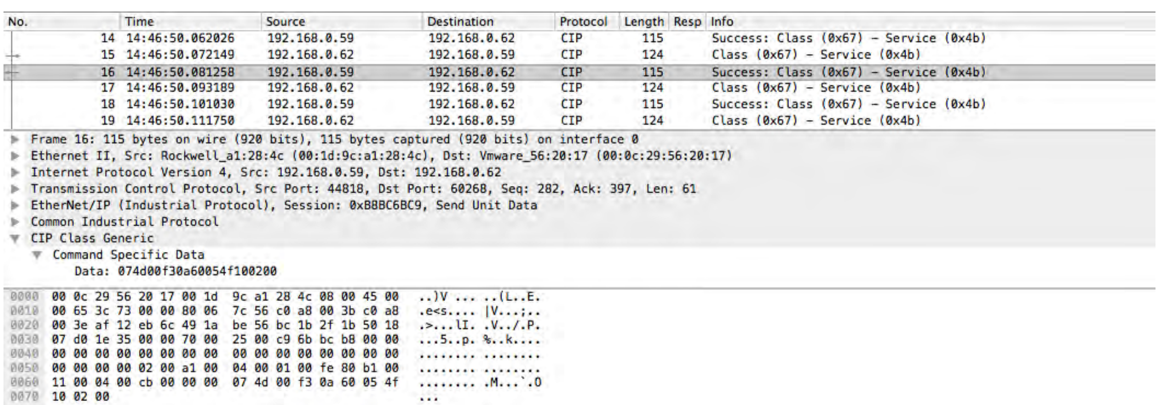
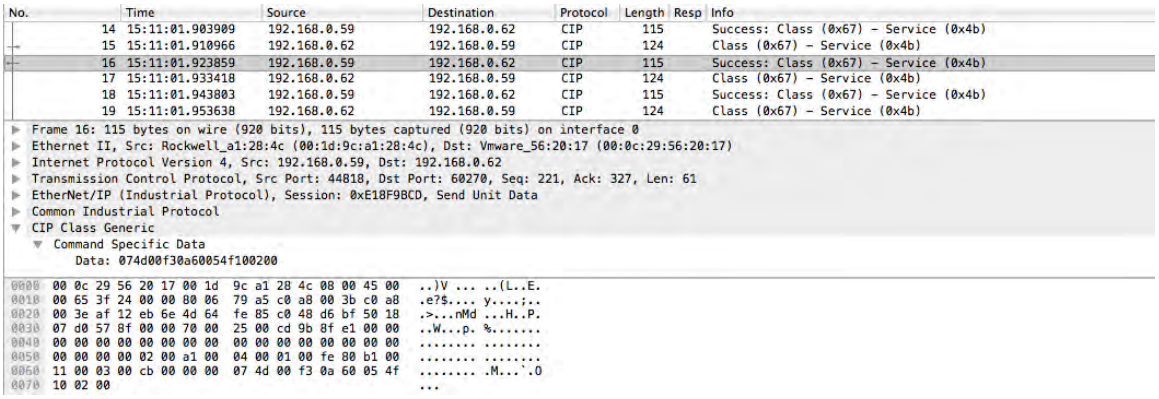


Figure 129. Protected Typed File Read Response over TCP (Size Fuzzed)

(2) T53 Results



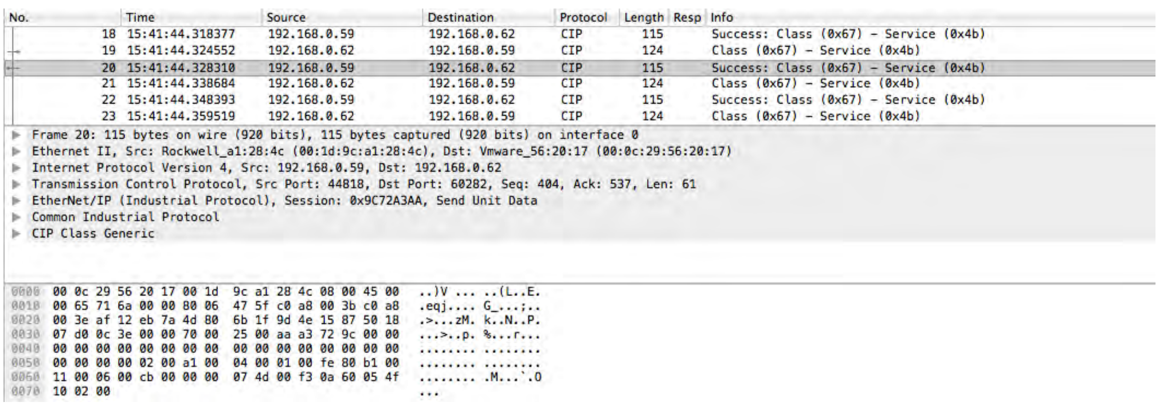
No.	Time	Source	Destination	Protocol	Length	Resp	Info
14	15:11:01.903909	192.168.0.59	192.168.0.62	CIP	115		Success: Class (0x67) - Service (0x4b)
15	15:11:01.910966	192.168.0.62	192.168.0.59	CIP	124		Class (0x67) - Service (0x4b)
16	15:11:01.923859	192.168.0.59	192.168.0.62	CIP	115		Success: Class (0x67) - Service (0x4b)
17	15:11:01.933418	192.168.0.62	192.168.0.59	CIP	124		Class (0x67) - Service (0x4b)
18	15:11:01.943803	192.168.0.59	192.168.0.62	CIP	115		Success: Class (0x67) - Service (0x4b)
19	15:11:01.953638	192.168.0.62	192.168.0.59	CIP	124		Class (0x67) - Service (0x4b)

```
Frame 16: 115 bytes on wire (920 bits), 115 bytes captured (920 bits) on interface 0
Ethernet II, Src: Rockwell_a1:28:4c (00:1d:9c:a1:28:4c), Dst: Vmware_56:20:17 (00:0c:29:56:20:17)
Internet Protocol Version 4, Src: 192.168.0.59, Dst: 192.168.0.62
Transmission Control Protocol, Src Port: 44818, Dst Port: 60270, Seq: 221, Ack: 327, Len: 61
EtherNet/IP (Industrial Protocol), Session: 0xE18F9BCD, Send Unit Data
Common Industrial Protocol
CIP Class Generic
  Command Specific Data
    Data: 074d00f30a60054f100200

0000  00 0c 29 56 20 17 00 1d  9c a1 28 4c 08 00 45 00   ..)V .... ..(L..E.
0010  00 65 3f 24 00 00 80 06  79 a5 c0 a8 00 3b c0 a8   .e?$.... y....;..
0020  00 3e af 12 eb 6e 4d 64  fe 85 c0 48 d6 bf 50 18   .>...nMd ...H..P.
0030  07 d0 57 8f 00 00 70 00  25 00 cd 9b 8f e1 00 00   ..W...p. %.......
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0050  00 00 00 00 02 00 a1 00  04 00 01 00 fe 80 b1 00   ........ ........
0060  11 00 03 00 cb 00 00 00  07 4d 00 f3 0a 60 05 4f   ........ .M...`.O
0070  10 02 00                                           ...
```

Figure 130. Protected Typed File Read Response over TCP (Tag Fuzzed)

(3) T54 Results



No.	Time	Source	Destination	Protocol	Length	Resp	Info
18	15:41:44.318377	192.168.0.59	192.168.0.62	CIP	115		Success: Class (0x67) - Service (0x4b)
19	15:41:44.324552	192.168.0.62	192.168.0.59	CIP	124		Class (0x67) - Service (0x4b)
20	15:41:44.328310	192.168.0.59	192.168.0.62	CIP	115		Success: Class (0x67) - Service (0x4b)
21	15:41:44.338684	192.168.0.62	192.168.0.59	CIP	124		Class (0x67) - Service (0x4b)
22	15:41:44.348393	192.168.0.59	192.168.0.62	CIP	115		Success: Class (0x67) - Service (0x4b)
23	15:41:44.359519	192.168.0.62	192.168.0.59	CIP	124		Class (0x67) - Service (0x4b)

```
Frame 20: 115 bytes on wire (920 bits), 115 bytes captured (920 bits) on interface 0
Ethernet II, Src: Rockwell_a1:28:4c (00:1d:9c:a1:28:4c), Dst: Vmware_56:20:17 (00:0c:29:56:20:17)
Internet Protocol Version 4, Src: 192.168.0.59, Dst: 192.168.0.62
Transmission Control Protocol, Src Port: 44818, Dst Port: 60282, Seq: 404, Ack: 537, Len: 61
EtherNet/IP (Industrial Protocol), Session: 0x9C72A3AA, Send Unit Data
Common Industrial Protocol
CIP Class Generic

0000  00 0c 29 56 20 17 00 1d  9c a1 28 4c 08 00 45 00   ..)V .... ..(L..E.
0010  00 65 71 6a 00 00 80 06  47 5f c0 a8 00 3b c0 a8   .eqj.... G_...;..
0020  00 3e af 12 eb 7a 4d 80  6b 1f 9d 4e 15 87 50 18   .>...zM. k..N..P.
0030  07 d0 0c 3e 00 00 70 00  25 00 aa a3 72 9c 00 00   ...>..p. %...r...
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0050  00 00 00 00 02 00 a1 00  04 00 01 00 fe 80 b1 00   ........ ........
0060  11 00 06 00 cb 00 00 00  07 4d 00 f3 0a 60 05 4f   ........ .M...`.O
0070  10 02 00                                           ...
```

Figure 131. Protected Typed File Read Response over TCP (Offset Fuzzed)

(4) T55 Results

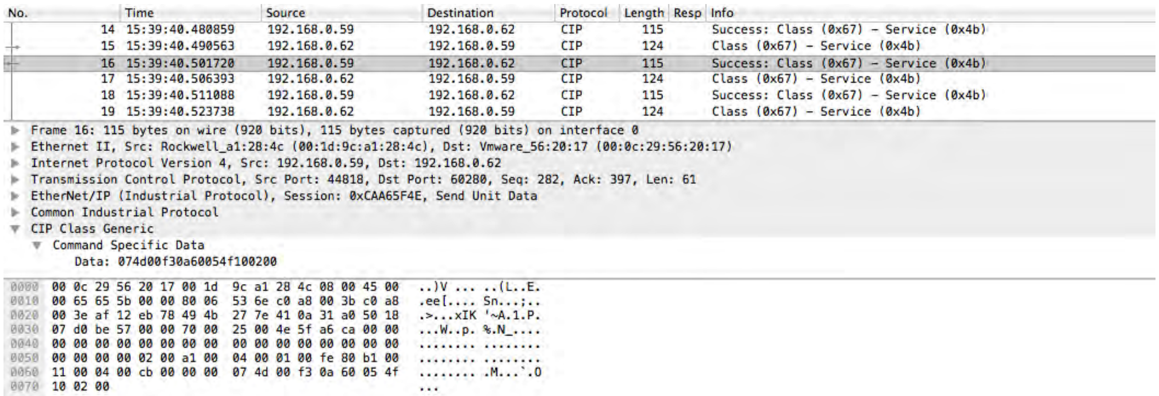


Figure 132. Protected Typed File Read Response over TCP (File Type Fuzzed)

C. PCCC PROTECTED TYPED FILE WRITE TEST CASES

This section shows the results of the PCCC Protected Typed File Write test cases.

(1) T56 Results

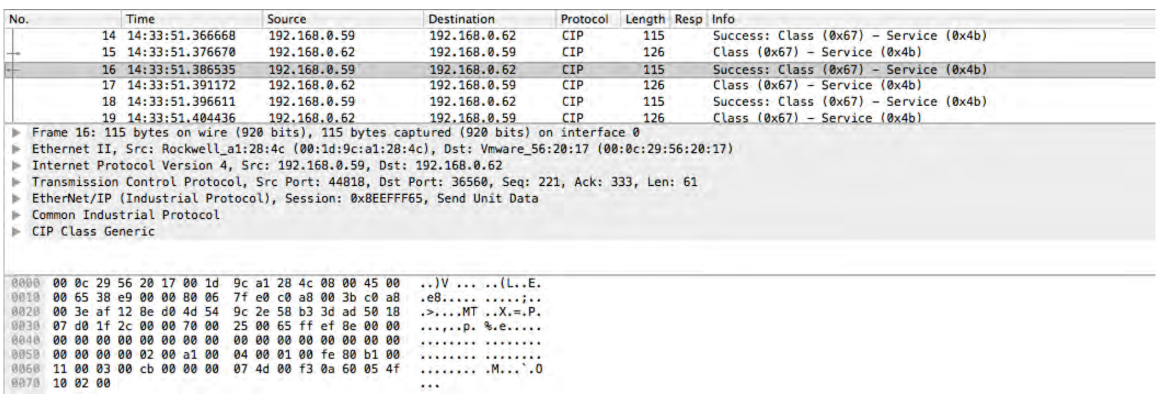


Figure 133. Protected Typed File Write Response over TCP (Size Fuzzed)

(2) T57 Results

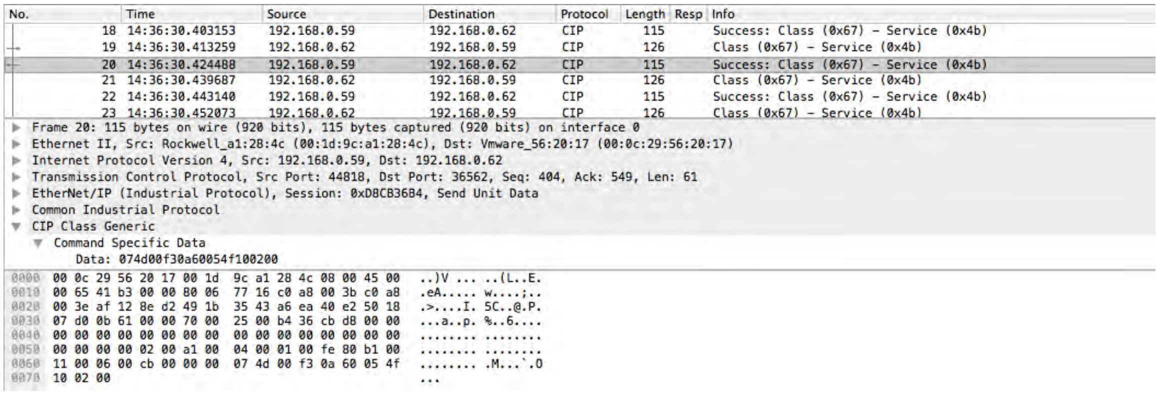


Figure 134. Protected Typed File Write Response over TCP (Tag Fuzzed)

(3) T58 Results

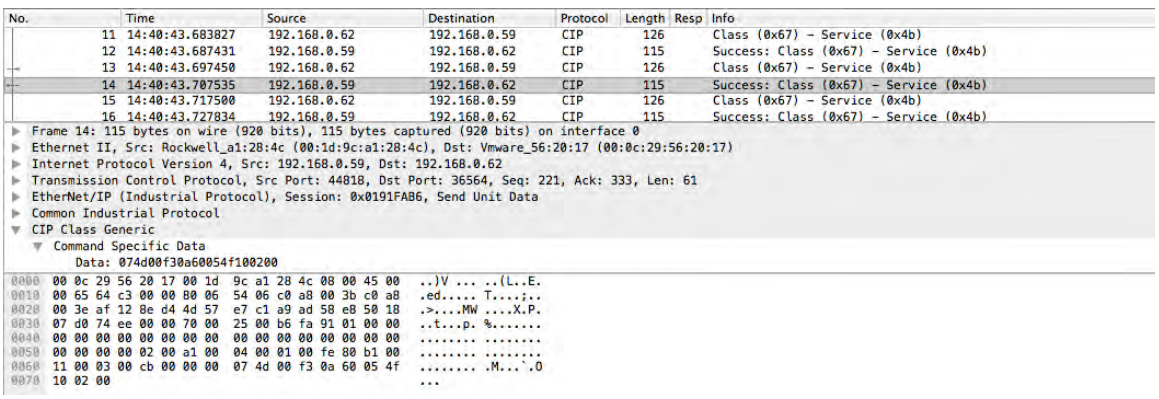


Figure 135. Protected Typed File Write Response over TCP (Offset Fuzzed)

(4) **T59 Results**

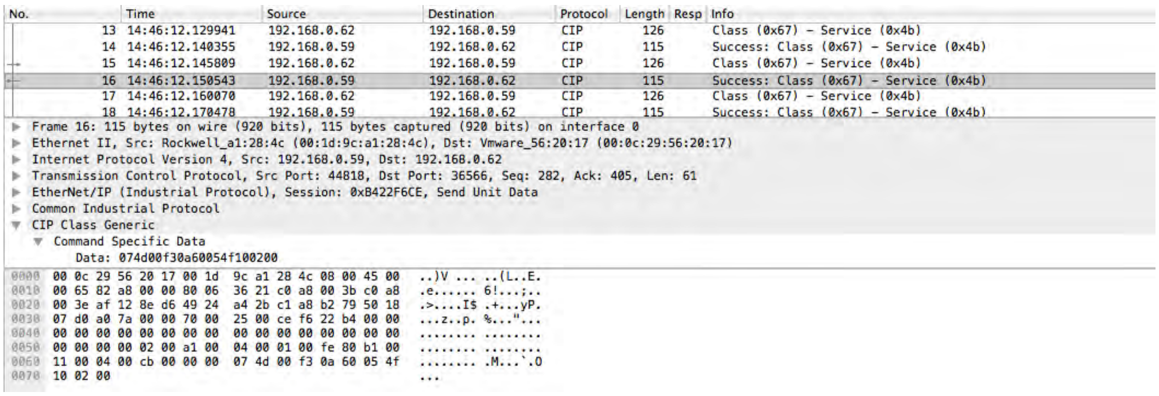


Figure 136. Protected Typed File Write Response over TCP (File Type Fuzzed)

(5) **T60 Results**

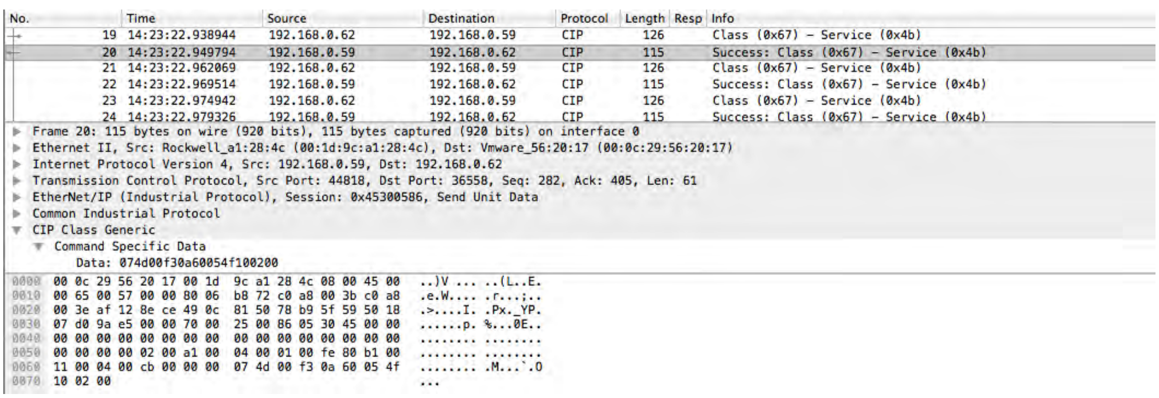


Figure 137. Protected Typed File Write Response over TCP (Data Fuzzed)

D. PCCC PROTECTED LOGICAL WRITE WITH THREE ADDRESS FIELDS TEST CASES

This section shows the results of the PCCC Protected Logical Write with Three Address Fields test cases.

(1) **T61 Results**

The Protected Logical Write with Three Address Fields responds with an EXT STS of 0x0B (“access denied, improper privilege”) when Byte Size is set to 0x00 (Figure

138). All other Byte Size inputs return responses with STS of 0x10 (“illegal command or format”) as demonstrated in Figure 139.

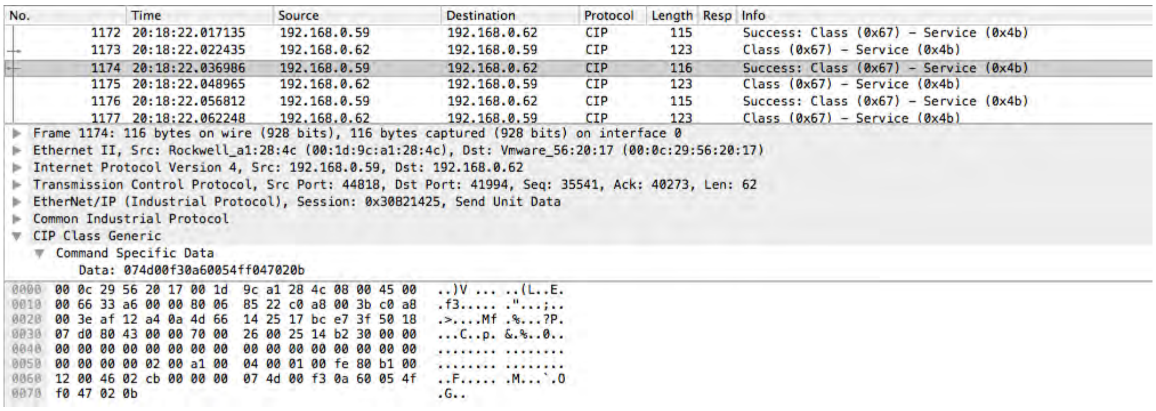


Figure 138. Protected Logical Write with Three Address Fields Response over TCP (Byte Size 0x00)

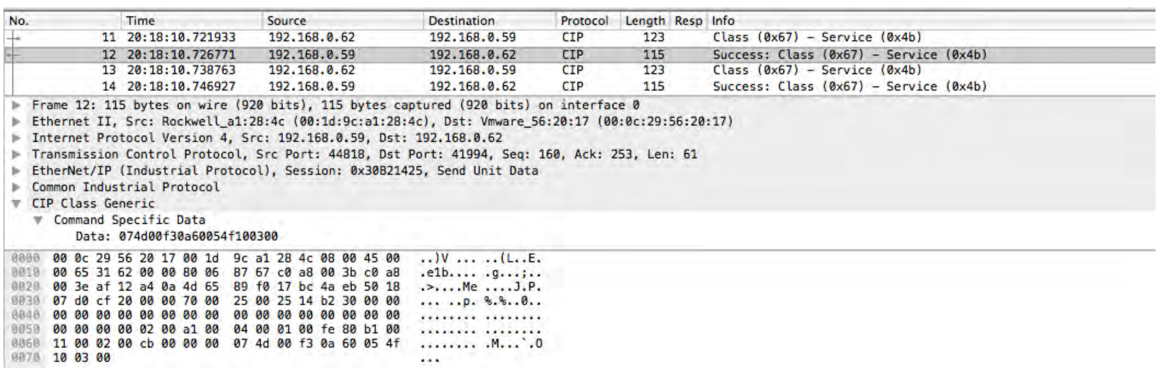


Figure 139. Protected Logical Write with Three Address Fields Response over TCP (Byte Size Fuzzed)

(2) T62 Results

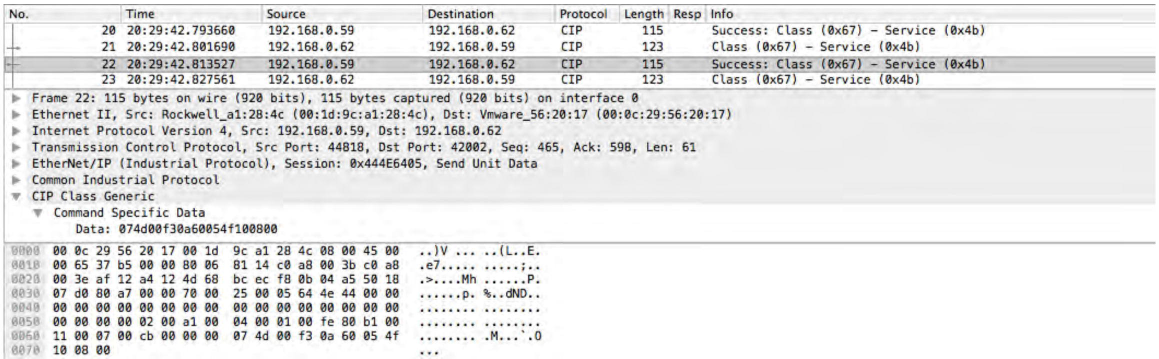


Figure 140. Protected Logical Write with Three Address Fields Response over TCP (File No. Fuzzed)

(3) T63 Results

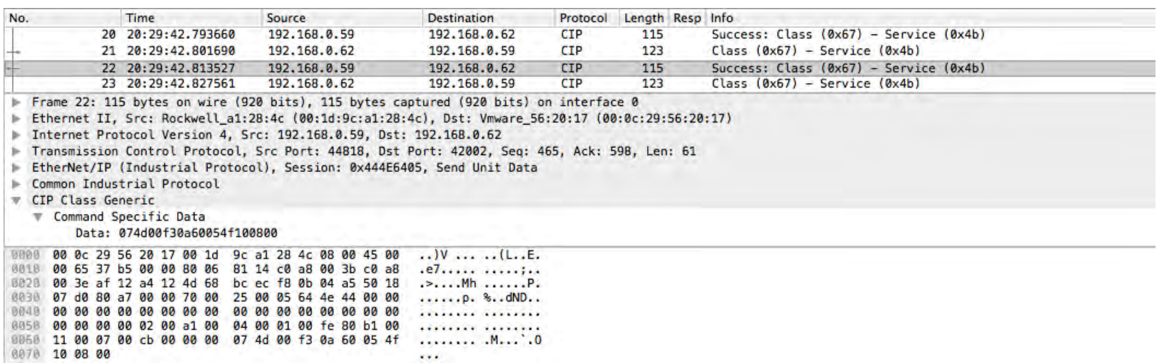


Figure 141. Protected Logical Write with Three Address Fields Response over TCP (File Type Fuzzed)

(4) T64 Results

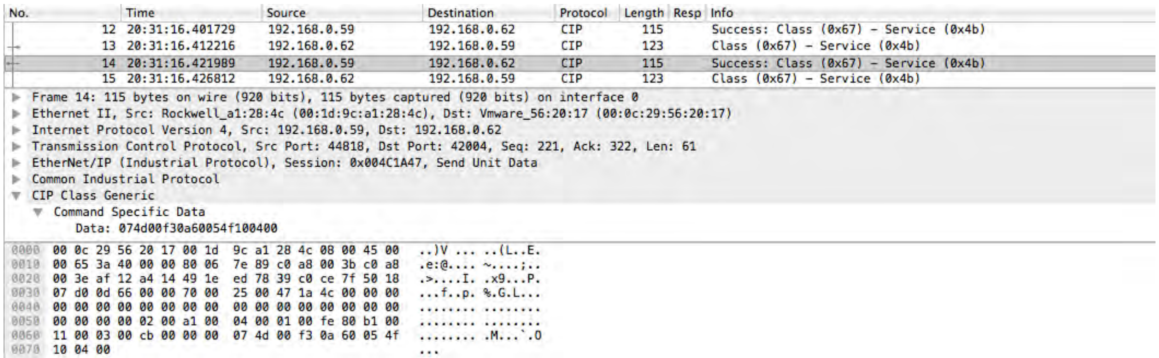


Figure 142. Protected Logical Write with Three Address Fields Response over TCP (Element No. Fuzzed)

(5) T65 Results

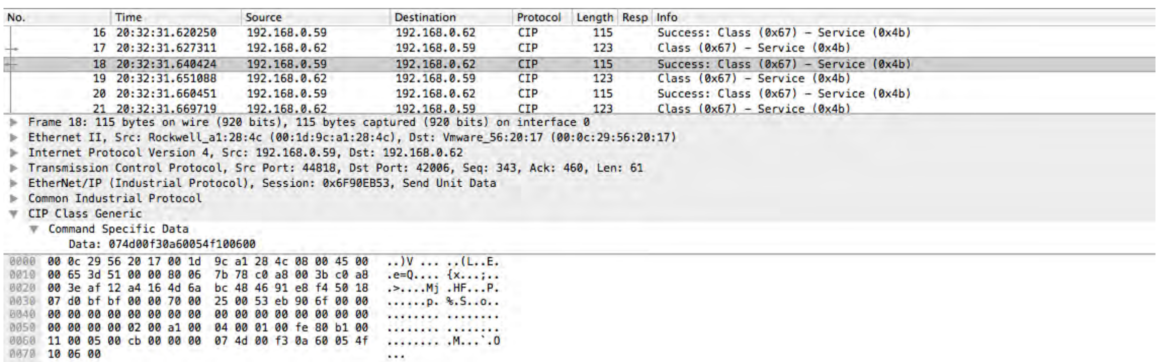


Figure 143. Protected Logical Write with Three Address Fields Response over TCP (Sub-Element No. Fuzzed)

E. PCCC UNPROTECTED READ TEST CASES

This section shows the results of the PCCC Unprotected Read test cases.

(1) T66 Results

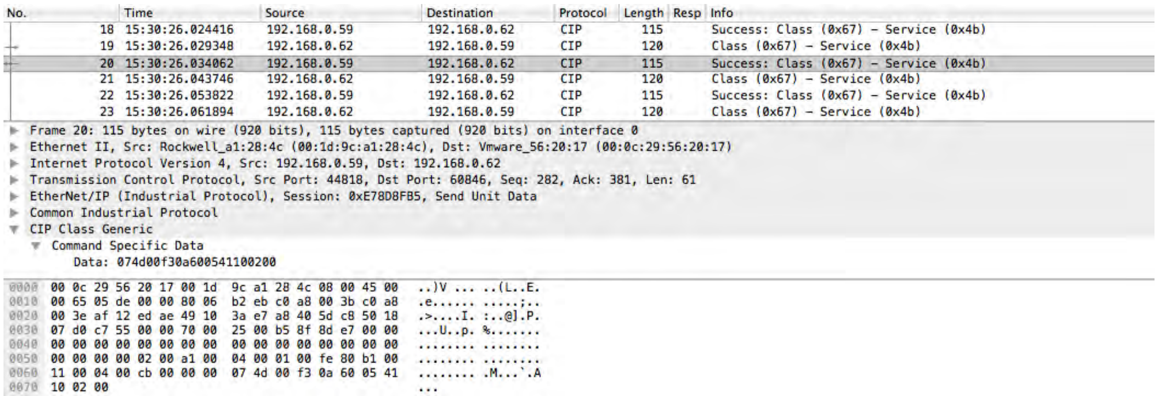


Figure 144. Unprotected Read Response over TCP (Address Fuzzed)

(2) T67 Results

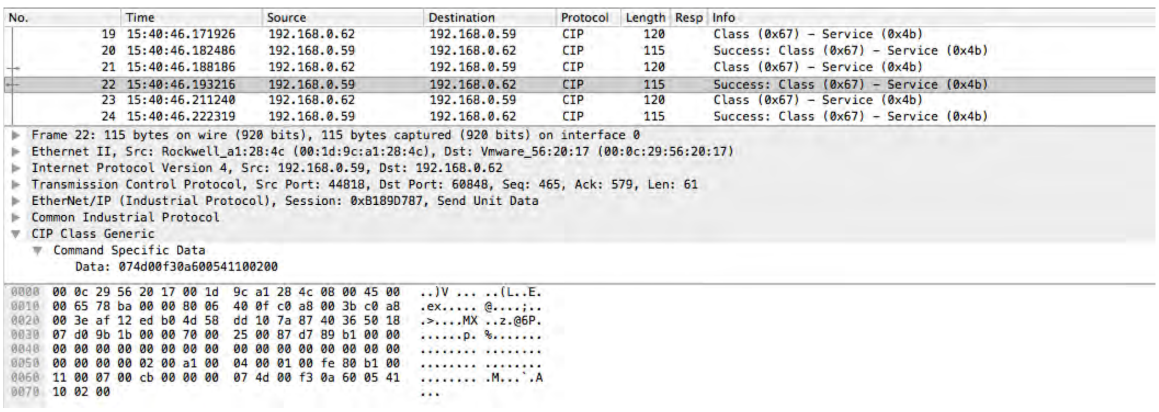


Figure 145. Unprotected Read Response over TCP (Size Fuzzed).

F. PCCC DIAGNOSTIC STATUS TEST CASES

This section shows the results of the PCCC Diagnostic Status test cases.

(1) T68 Results

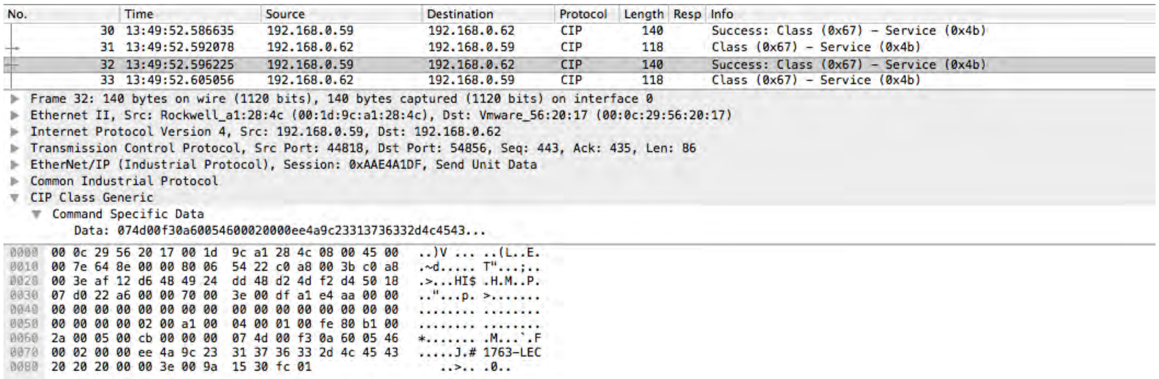


Figure 146. Diagnostic Status Response over TCP (Functionality Test)

G. PCCC READ DIAGNOSTIC COUNTERS TEST CASES

This section shows the results of the PCCC Read Diagnostic Counters test cases.

(1) T69 Results

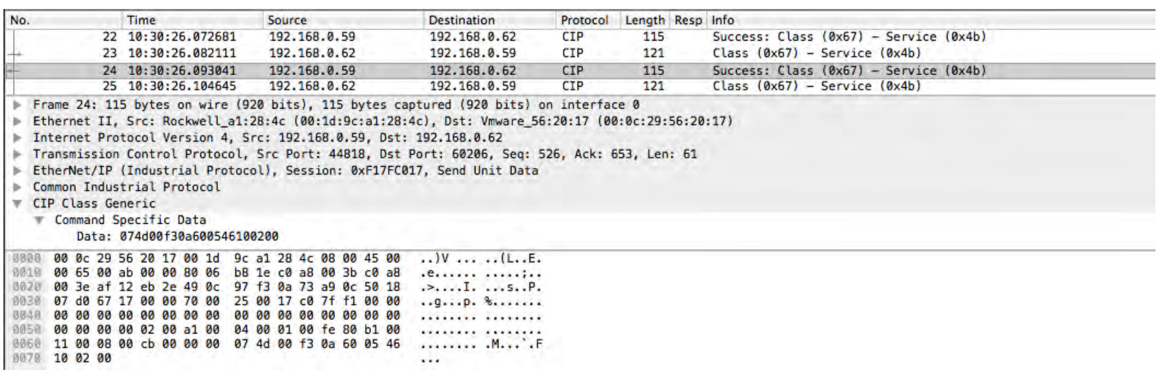


Figure 147. Read Diagnostic Counters Response over TCP
(Address Fuzzed: 0x3455)

(2) T70 Results

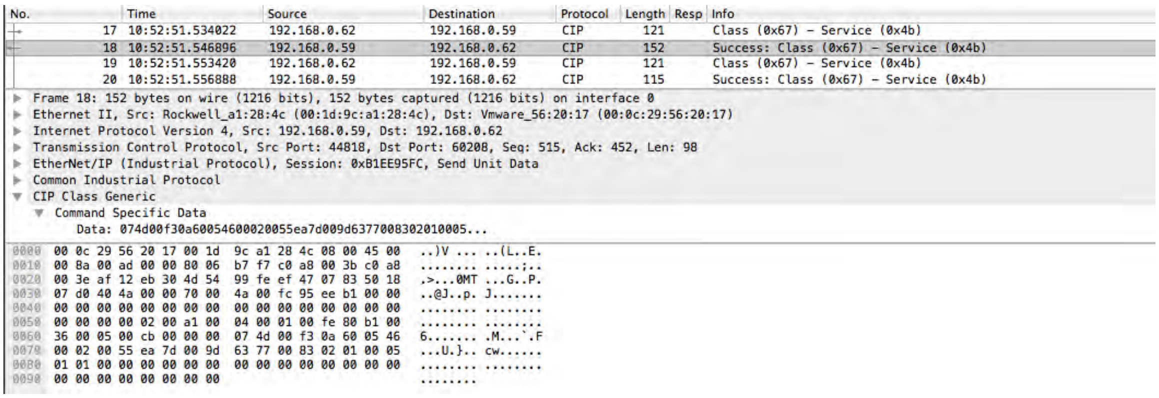


Figure 148. Read Diagnostic Counters Response over TCP (Size Fuzzed: 25)

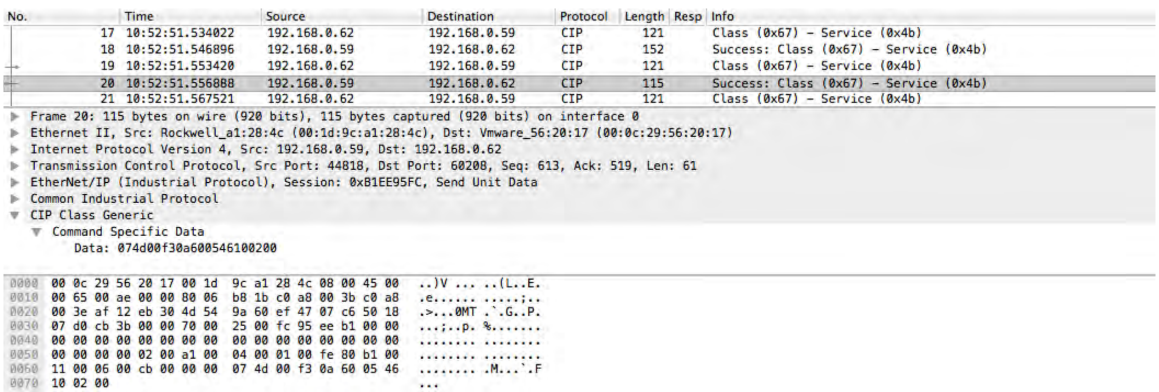


Figure 149. Read Diagnostic Counters Response over TCP (Size Fuzzed: 75)

H. PCCC RESTART TEST CASES

This section shows the results of the PCCC Restart test cases.

(1) T71 Results

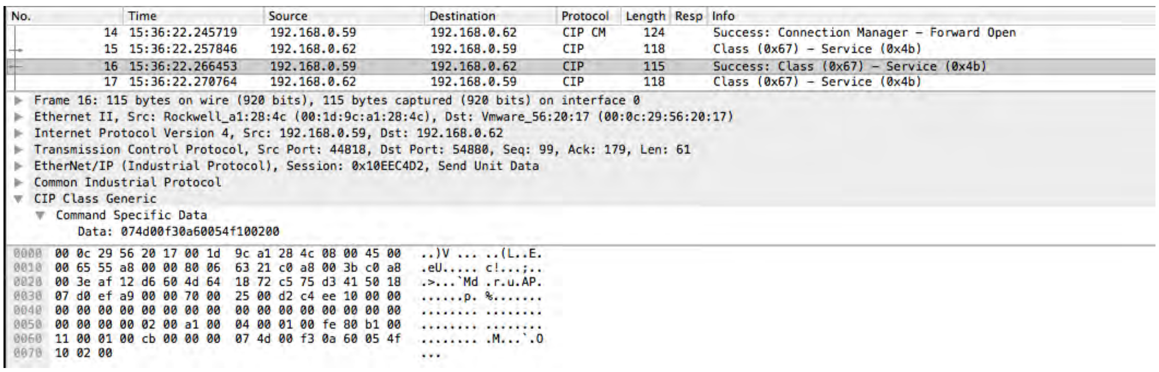


Figure 150. Restart Response over TCP (Functionality Test)

I. PCCC DOWNLOAD COMPLETED TEST CASES

This section shows the results of the PCCC Download Completed test cases.

(1) T72 Results

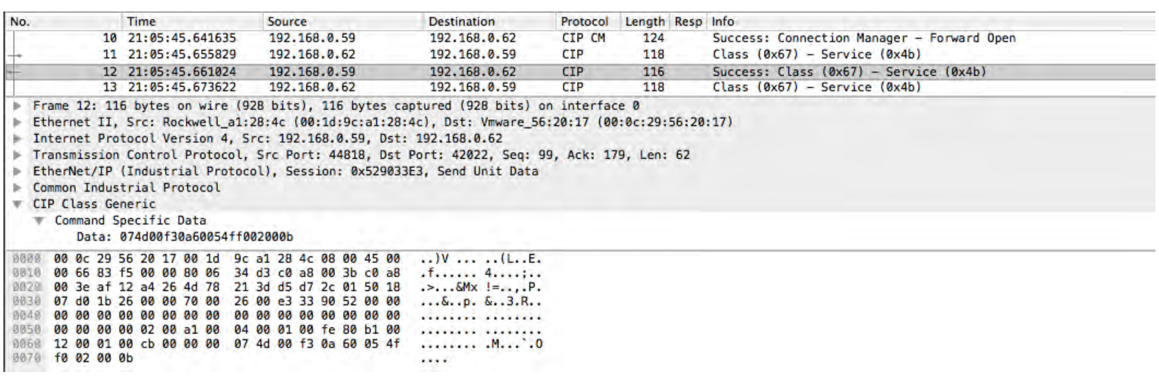


Figure 151. Download Completed Response over TCP (Functionality Test)

J. PCCC PROTECTED LOGICAL READ WITH THREE ADDRESS FIELDS TEST CASES

(1) T73 Results

For comparison, Figure 152 illustrates a Protected Logical Read with Three Address Fields request packet with File No. 0x03 and File Type 0x47 field inputs sent to a MicroLogix 1100 PLC. The SUT enters a fault state upon receiving the packet, i.e., no CIP response is observed. Figure 153 illustrates a similar request with identical File No. and File Type fields sent to the ControlLogix PLC. Figure 154 displays the ControlLogix PLC's response to the test packet. The ControlLogix does not fault. The response packet contains an EXT STS code of 0x06.

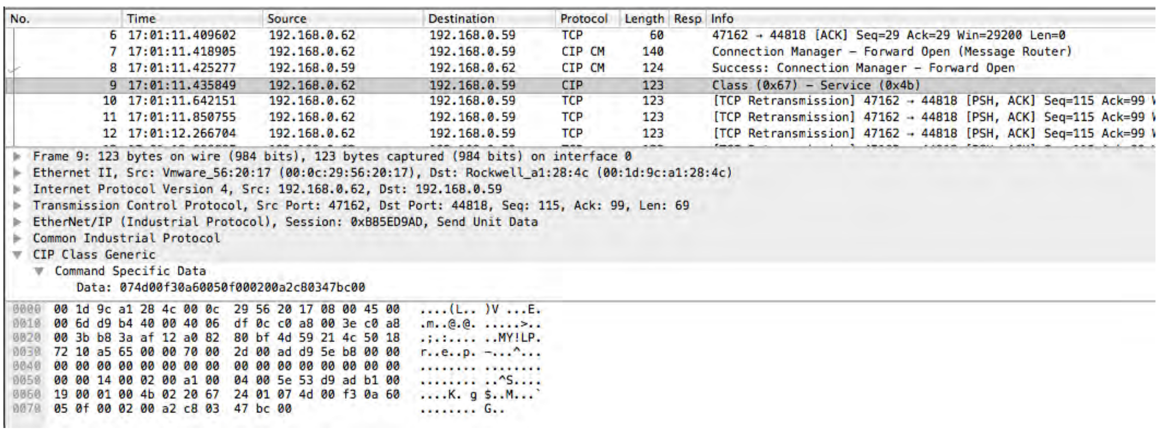


Figure 152. MicroLogix Protected Logical Read with Three Address Fields Request over TCP (File No. 0x03 and File Type 0x47)

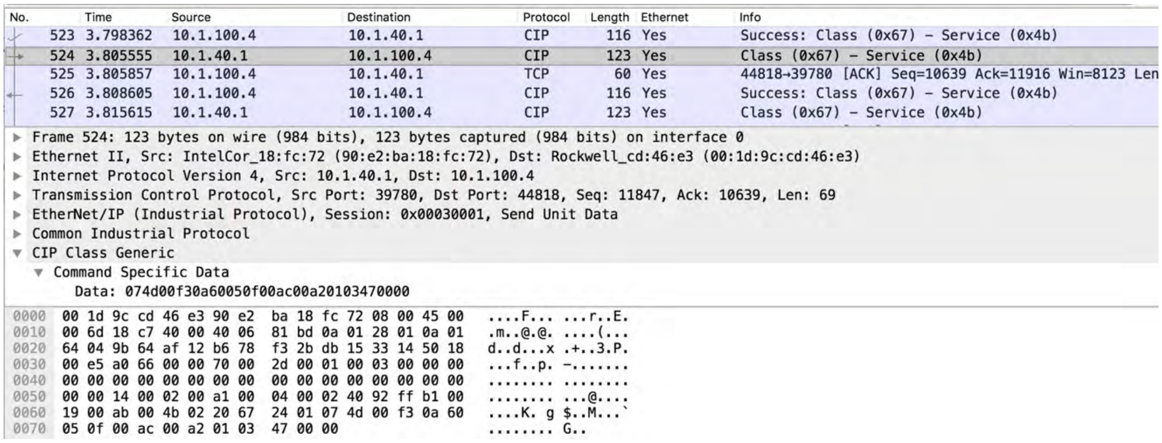


Figure 153. ControlLogix Protected Logical Read with Three Address Fields Request over TCP (File No. 0x03 and File Type 0x47)

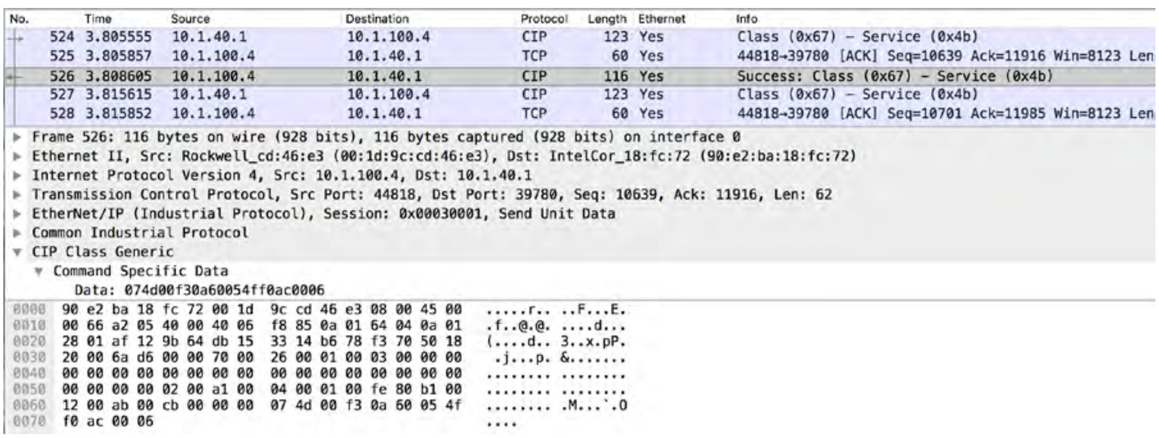


Figure 154. ControlLogix Protected Logical Read with Three Address Fields Response over TCP (File No. 0x03 and File Type 0x47)

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] J. Slay and M. Michael, "Lessons learned from the Maroochy Water breach," in *Critical Infrastructure Protection*, Boston, MA: Springer, 2008, pp. 73–82.

[2] J. R. Lindsay, "Stuxnet and the limits of cyber warfare," *Security Studies*, vol. 22, pp. 365–404, 2013.

[3] K. Zetter, "Inside the cunning, unprecedented hack of Ukrain's power grid," March 3, 2016. [Online]. Available: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/

[4] S. Young, "Dallas siren hack came via radio frequency, not computer, city says," April 10, 2017. [Online]. Available: www.dallasobserver.com/news/dallas-siren-hack-done-by-radio-not-computer-9358087

[5] M. Swearingen, S. Brunasso, J. Weiss and D. Huber, "What you need to know (and don't) about the AURORA vulnerability," *Power Magazine*, September 1, 2013. [Online]. Available: http://www.powermag.com/what-you-need-to-know-and-dont-about-the-aurora-vulnerability/?pagenum=2

[6] J. Mulder, M. Schwartz, M. Berg, J. R. Van Houten, J. M. Urrea, M. A. King, A. A. Clements and J. Jacob, "WeaselBoard: Zero-day exploit detection for programmable logic controllers," Albuquerque, NM, 2013.

[7] D. G. Peterson, "Digital bond," January 19, 2012. [Online]. Available: http://www.digitalbond.com/blog/2012/01/19/project-basecamp-at-s4/

[8] B. P. Miller, L. Fredriksen and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM 33,* vol. 12, December 1990.

[9] F. Tacliad, "ENIP fuzz: A scapy-based EtherNet/IP fuzzer for security testing," M.S. thesis, Dept. of Computer Science., NPS, Monterey, CA, USA, 2016.

[10] M. Henry, M. Iacovelli and J. Thatcher, "DDG 1000 engineering control system (ECS)." [Online]. Available: https://seagrant.mit.edu/ESRDC_library/Henry-DDG-1000.pdf

[11] ICS-CERT, "Ongoing sophisticated malware campaign compromising ICS (update E)," 2014. [Online]. Available: https://ics-cert.us-cert.gov/alerts/ICS-ALERT-14-281-01B

[12] B. Freeman, "A new defense for Navy ships: Protection from cyber attacks," September 17, 2015. [Online]. Available: https://www.onr.navy.mil/en/Media-Center/Press-Releases/2015/RHIMES-Cyber-Attack-Protection.aspx

[13] R. Grandgenett, R. Gandhi and W. Mahoney, “Exploitation of Allen Bradley’s implementation of EtherNet/IP for denial of service against industrial control systems,” in *9th International Conference on Cyber Warfare and Security, Purdue University*, 2014.

[14] R. Grandgenett, W. Mahoney, and R. Gandhi, “Authentication bypass and remote escalated I/O command attacks,” in *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, 2015.

[15] ODVA & ControlNet International, Ltd., *The CIP Networks Library. Vol. 1. Common Industrial Protocol (CIP)*, 3.22 ed., Ann Arbor, MI: Open DeviceNet Vendor Association, Inc., 2017.

[16] ODVA & ControlNet International, Ltd., *The CIP Networks Library Volume 2: EtherNet/IP Adaptation of CIP*, 1.23 ed., Ann Arbor, MI: Open DeviceNet Vendor Association, Inc., 2017.

[17] Allen Bradley, *DF1 Protocol and Command Set Reference Manual*, Milwaukee, WI, USA, October 1996.

[18] SECDEV, “Scapy,” accessed April 3, 2017. [Online]. Available: http://www.secdev.org/projects/scapy/

[19] P. Brooks, “EtherNet/IP: Industrial protocol white paper,” October 2001. [Online]. Available: http://literature.rockwellautomation.com/idc/groups/literature/documents/wp/enet-wp001_-en-p.pdf

[20] V Schiffer on behalf of Rockwell Automation, “The Common Industrial Protocol (CIP) and the Family of CIP Networks,” ODVA, Inc., Ann Arbor, MI, USA, 2016.

[21] Rockwell Automation, “Communicating with RA Products Using EtherNet/IP Explicit Messaging (Rev 1.2),” Rockwell Autmation, 2001. [Online]. Available: https://www.rockwellautomation.com/resources/downloads/rockwellautomation/pdf/sales-partners/technology-licensing/eipexp1_2.pdf

[22] ODVA, “The Common Industrial Protocol (CIP),” 2017. [Online]. Available: https://www.odva.org/Technology-Standards/Common-Industrial-Protocol-CIP/Overview

[23] ODVA, “Common Industrial Protocol (CIP),” 2006. [Online]. Available: https://scadahacker.com/library/Documents/ICS_Protocols/ODVA%20-%20CIP.pdf

[24] Rockwell Automation, “Delivery of CIP over RA Serial DF1 Links (Rev. 1.1),” Rockwell Automation, 2006. [Online]. Available: https://www.rockwellautomation.com/resources/downloads/rockwellautomation/pdf/sales-partners/technology-licensing/CIPandPCCC_v1_1.pdf

[25] N. Champey, “Principles of EtherNet/IP Communication,” accessed April 24, 2017. Rueil-Malmaison, FR. [Online]. Available: https://scadahacker.com/library/Documents/ICS_Protocols/Schneider%20-%20Principles%20of%20EtherNetIP%20Communication.pdf

[26] Allen-Bradley / Rockwell Automation, “Enhanced DeviceNet communications module,” May 2000. Milwaukee, WI, USA. [Online]. Available: http://literature.rockwellautomation.com/idc/groups/literature/documents/um/1203-um014_-en-p.pdf

[27] R. Shapiro, S. Bratus, E. Rogers, and S. Smith, “Do-it-yourself SCADA vulnerability testing with LZFuzz,” accessed April 15, 2017. [Online]. Available: http://www.cs.dartmouth.edu/~sws/pubs/sbs11.pdf

[28] D. Aitel, “The advantages of block-based protocol analysis for security testing,” February 4, 2002. [Online]. Available: http://www.immunitysec.com/downloads/advantages_of_block_based_analysis.pdf

[29] Beyond Security, “Dynamic testing (fuzzing) on the Ethernet IP Protocol by beSTORM Ethernet IP with beSTORM,” accessed April 25, 2017. [Online]. Available: http://www.beyondsecurity.com/dynamic_fuzzing_testing_ethernet_IP_protocol

[30] A. Portnoy, P. Amini and R. Sears. “Sulley,” accessed April 20, 2017. [Online]. Available: https://github.com/OpenRCE/sulley

[31] S. Bansal and N. Bansal, “Scapy—A Python tool for security testing,” *Journal of Computer Science & Systems Biology*, March 31, 2015.

[32] P. Joshi, P. Patel, and R. Parikh, “A fuzz testing framework for Wi-Fi devices,” *International Journal of Research in Engineering & Advanced Technology*, vol. 3, no. 5, October-November 2015.

[33] H. Rafiee, C. Mueller, L. Niemeier, J. Streek, C. Sterz, and C. Meinel, “A flexible framework for detecting IPv6 vulnerabilities,” in *The 6th International Conference on Security of Information Networks*, Aksaray, Turkey, 2013.

[34] A. Lahmadi, C. Bernardini, and O. Festor, “A testing framework for discovering vulnerabilities in 6LoWPAN networks,” in *8th International Conference on Distributed Computing in Sensor Systems (DCOSS2012)*, Hangzhou, China., 2012.

[35] P. Tsankov, T. Dashti, and D. Basin, “SECFUZZ: Fuzz-testing security protocols,” in *7th International Workshop on Automation of Software Test*, Zurich, Switzerland, 2012.

[36] A. G. Voyiatzis, K. Katsigiannis, and S. Koubias, *A Modbus/TCP Fuzzer for Testing Internetworked Industrial Systems*, IEEE, 2015.

[37] MODICON, Inc., Industrial Automation Systems, *Modicon Modbus Protocol Reference Guide*, North Andover, MA: MODICON, Inc., Industrial Automation Systems, 1996.

[38] R. Sprabery, T. Morris, and S. P. V. Madani, “Protocol mutation intrusion detection for synchrophasor communications,” in *Eighth Annual Cyber Security and Information Intelligence Research Workshop*, Oak Ridge, TN, 2012.

[39] Rockwell Automation, “MicroLogix Programmable Controllers Selection Guide,” Milwaukee, WI, USA, 2015. Accessed January 24, 2017. [Online]. Available: http://literature.rockwellautomation.com/idc/groups/literature/documents/sg/1761-sg001_-en-p.pdf

[40] Allen-Bradley Rockwell Automation, “ControlLogix 5570 Controllers,” accessed January 23, 2017. [Online]. Available: http://ab.rockwellautomation.com/Programmable-Controllers/ControlLogix/5570-Controllers#overview

[41] ICS-CERT, “Advisory (ICSA-17-138-03) Rockwell Automation MicroLogix 1100 Controllers,” July 18, 2017. [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-17-138-03

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California